

LNCS 2065

**Herman Balsters
Bert de Brock
Stefan Conrad (Eds.)**

Database Schema Evolution and Meta-Modeling

**9th International Workshop on Foundations of Models and
Languages for Data and Objects, FoMLaDO/DEMM 2000
Dagstuhl Castle, Germany, September 2000, Selected Papers**



Springer

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

2065

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Herman Balsters Bert de Brock
Stefan Conrad (Eds.)

Database Schema Evolution and Meta-Modeling

9th International Workshop on Foundations
of Models and Languages for Data and Objects
FoMLaDO/DEMM 2000
Dagstuhl Castle, Germany, September 18-21, 2000
Selected Papers



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Herman Balsters
University of Twente, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail:balsters@cs.utwente.nl

Bert de Brock
University of Groningen, Faculty of Management and Organization
P.O. Box 800, 9700 AV Groningen, The Netherlands
E-mail:e.o.de.brock@bdk.rug.nl

Stefan Conrad
University of Munich, Department of Computer Science
Oettingenstraße 67, 80538 Munich, Germany
E-mail:conrad@informatik.uni-muenchen.de

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Database schema evolution and meta modeling : selected papers / 9th
International Workshop on Foundations of Models and Languages for Data and
Objects, FoMLaDO DEMM 2000, Dagstuhl Castle, Germany, September 18 - 21,
2000. Herman Balsters ... (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo :
Springer, 2001
(Lecture notes in computer science ; Vol. 2065)
ISBN 3-540-42272-2

CR Subject Classification (1998): H.2, H.3

ISSN 0302-9743

ISBN 3-540-42272-2 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10781739 06/3142 5 4 3 2 1 0

Preface

The Ninth International Workshop on Foundations of Models and Languages for Data and Objects (FoMLaDO) took place in Dagstuhl Germany, September 18–21, 2000. The topic of this workshop was Database schema Evolution and Meta-Modeling; this FoMLaDO Workshop was hence assigned the acronym DEMM 2000.

These post-proceedings contain the revised versions of the accepted papers of the DEMM 2000 workshop. Twelve regular papers were accepted for inclusion in the proceedings. The papers address the following issues:

- Consistency of evolving concurrent information systems
- Adaptive specifications of technical information systems
- Change propagation in schema evolution of object-based systems
- Evolving software of a schema evolution system
- Logical characterization of schema evolution
- Conflict management in integrated databases
- Evolving relation schemas
- Conceptual descriptions of adaptive information systems
- OQL-extensions for metadata access
- Metamodeling of schema evolution
- Metrics for conceptual schema evolution
- Incremental datawarehouse construction

In addition to the regular papers, there is an invited paper by Can Türker on schema evolution in SQL99 and (object-)relational databases.

Acknowledgements: We wish to thank the program committee members for their work on reviewing the submitted papers. We also wish to thank all authors for submitting papers to this workshop. Moreover, all participants of the workshop are thanked for contributing to lively discussions. Thanks also to Elke Rundensteiner, who delivered an invited talk on the SERF-project concerning flexible database transformations. Finally, we wish to thank Gunter Saake and Can Türker for their initial help in organizing this workshop.

April 2001

Herman Balsters
Bert de Brock
Stefan Conrad

Program Committee

Herman Balsters	University of Twente, The Netherlands; Chair
Elisa Bertino	University of Milan, Italy
Anthony Bonner	University of Toronto, Canada
Bert de Brock	University of Groningen, The Netherlands; Co-chair
Jan van den Bussche	University of Limburg, Belgium
Stefan Conrad	University of Munich, Germany; Co-chair
Hans-Dieter Ehrich	TU Braunschweig, Germany
Paul Grefen	University of Twente, The Netherlands
Theo d'Hondt	University of Brussels, Belgium
Kalle Lyytinen	University of Jyväskylä, Finland
John Jules Meyer	University of Utrecht, The Netherlands
Michele Missikoff	IASI-CNR, Italy
Tamer Özsu	University of Alberta, Canada
Jan Paredaens	University of Antwerp, Belgium
Oscar Pastor	Technical University of Valencia, Spain
Gunter Saake	University of Magdeburg, Germany
Amilcar Sernadas	IST Lisbon, Portugal
Klaus-Dieter Schewe	TU Clausthal, Germany
Marc Scholl	University of Konstanz, Germany
Can Türker	ETH Zurich, Switzerland
Jari Veijalainen	University of Jyväskylä, Finland
Gottfried Vossen	University of Munster, Germany
Gerhard Weikum	University of the Saarland, Germany
Roel Wieringa	University of Twente, The Netherlands

Table of Contents

Invited Talk

Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS . . .	1
<i>Can Türker (Swiss Federal Institute of Technology (ETH) Zurich)</i>	

Regular Papers

Consistency Management in Runtime Evolving Concurrent Information Systems: A CO-NETS-Based Approach	33
<i>Nasreddine Aoumeur, Gunter Saake (University of Magdeburg)</i>	
Adaptive Specifications of Technical Information Systems	61
<i>Sören Balko (University of Magdeburg)</i>	
Evolving the Software of a Schema Evolution System	68
<i>Kajal T. Claypool, Elke A. Rundensteiner, George T. Heineman (Worcester Polytechnic Institute)</i>	
Schema Evolution and Versioning: A Logical and Computational Characterisation	85
<i>Enrico Franconi (University of Manchester), Fabio Grandi, Federica Mandreoli (Università di Bologna)</i>	
Temporal Branching as a Conflict Management Technique	100
<i>Roy Gelbard, Asher Gilmour (Ben-Gurion University, Beer-Sheva)</i>	
Evolving Relations	115
<i>Ole G. Jensen, Michael H. Böhlen (Aalborg University)</i>	
QFD Matrix for Incremental Construction of a Warehouse via Data Marts	133
<i>Ron McFadyen, Fung-Yee Chan (University of Winnipeg)</i>	
Change Propagation in an Axiomatic Model of Schema Evolution for Objectbase Management Systems	142
<i>Randal J. Peters (University of Manitoba), Ken Barker (University of Calgary)</i>	
Evolving Objects: Conceptual Description of Adaptive Information Systems	163
<i>Gunter Saake (University of Magdeburg), Can Türker (Swiss Federal Institute of Technology (ETH) Zurich), Stefan Conrad (University of Munich)</i>	

Extending the Object Query Language for Transparent Metadata Access . . 182
Hong Su, Kajal T. Claypool, Elke A. Rundensteiner
(Worcester Polytechnic Institute)

A Metamodeling Approach to Evolution 202
Marie-Noëlle Terrasse (Université de Bourgogne, France)

Defining Metrics for Conceptual Schema Evolution 220
Lex Wedemeijer (ABP Pensioenen, The Netherlands)

Author Index 245

Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS

Can Türker

Swiss Federal Institute of Technology (ETH) Zurich
Institute of Information Systems, ETH Zentrum
CH-8092 Zurich, Switzerland
tuerker@inf.ethz.ch

1 Introduction

A *database schema* denotes the description of the structure and behavior of a database. Straightforwardly, (database) *schema evolution* refers to changes of the database schema that occur during the lifetime of the corresponding database. It particularly refers to changes of schema elements already stored in the database.

The information about a database schema is stored in the schema catalog. Data stored in these catalogs is referred to as meta-data. In this sense, schema evolution could be seen as a change of the content of the schema catalog.

In an object-relational database model, such as proposed in SQL-99 [Int99], a database schema, among others, consists of the following elements:

- types, tables, and views,
- subtype and subtable relationships,
- constraints and assertions,
- functions, stored procedures, and triggers, and
- roles and privileges.

Thus more precisely, schema evolution can be defined as the creation, modification, and removal of such kinds of **schema** elements. □

Although schema evolution is a well-known and partially well-studied topic, an overview and comparison of schema evolution language constructs provided in the SQL standard as well as in commercial database management systems is still missing. This survey paper intends to fill this gap. First, in Section 2, we give an overview of schema evolution operations supported by the new SQL standard, called SQL-99 [Int99]. Thereafter, in Section 3, we compare major commercial (object-)relational database management systems with respect to the support of these operations and others disregarded in SQL-99. Finally, we conclude the paper with some remarks on open schema evolution issues neglected in SQL-99 as well as in the current **implementations** of commercial database management systems.

2 Schema Evolution in SQL-99

Before we present the schema evolution language constructs provided in SQL-99 [Int99], we briefly introduce the basic notions and concepts of SQL-99.

2.1 Basic Schema Elements

The main concept for representing data in SQL-99 is the concept of a table, which is made up by a set of columns and rows. A *table* is associated with a schema and an instance:

- A *schema* of a table specifies the name of the table, the name of each column, and the domains (data types) associated with the columns. A domain is typically referred to by a domain name and has a set of associated values. Examples for basic domains (built-in data types) in SQL-99 are **INTEGER**, **REAL**, **NUMERIC**, **CHAR**, or **DATE**.
- An *instance* of a table schema, called *table*, is a set of rows where each row has the same structure as defined in the table schema, that is, each row the same number of columns and the values of the columns are taken from the corresponding domain.

A table is either a *base table* or a *derived table*. A derived table is a table that is derived from one or more other tables by the evaluation of a query expression. A *view* is a named derived table.

Besides the standard built-in data types, SQL-99 provides a *row* type constructor, an *array* type constructor and a *reference* type constructor. A row type constructor is used to define a column consisting of a number of *fields*. Any data type can be assigned to a field. The array type constructor is also applicable to any data type, whereas the applicability of the reference type constructor is restricted to user-defined types only.

A *user-defined type* is a *named* data type. SQL-99 distinguishes two kinds of user-defined types: (1) *distinct* types which are copies of predefined data types and (2) *structured* types which define a number of *attributes* and *method* specifications. Every attribute is associated with a data type, which itself can also be a user-defined type.

Structured types can be set into a subtype relationship. A *subtype* implicitly inherits the attributes and method specifications from its *supertype*. Every structured type may have at most one *direct* supertype. That is, SQL-99 does not support *multiple inheritance*.

A table that is created based on a structured type is called a *typed table*. Typed tables can be organized within a table hierarchy. A table can be a subtable of at most one direct supertable. All rows of a subtable are implicitly contained in all supertables of that table. Analogously to (base) tables, views can be typed and organized in view hierarchies.

A table column may rely on a built-in data type, row type, user-defined type, reference type, or collection type. The same holds for an attribute of a structured type.

SQL-99 furthermore supports the following concepts:

- *Domains* are named sets of values that are associated with a default value and a set of domain constraints.
- *Assertions* are named constraints that may relate to the content of individual rows of a table, to the entire content of a table, or to the contents of more than one table.

- *Routines* (*procedures* and *functions*) and *triggers* are named execution units that are used to implement application logic in the database.
- *Roles* and *privileges* are used to implement a security model. A role is a named group of related privileges which can be granted to users or roles.

To sum up, domains, user-defined types, tables, views, assertions, routines (procedures and functions), triggers, roles, and privileges are the basic schema elements in SQL-99. A *database schema* is formed by a set of schema element definitions and it evolves by adding, altering, or removing schema element definitions. It is important to note that some schema evolution operations may also have an effect on the actual database objects, for instance, on the rows of a table. In the following, we will see which language constructs are provided in SQL-99 to evolve a database schema. Before, to give an overview of the available operations, we summarize the main schema evolution operations in Table 1.

Table 1. Main Operations of Schema Evolution in SQL-99

	CREATE	ALTER	DROP
DOMAIN	✓	✓	✓
TYPE	✓	✓	✓
TABLE	✓	✓	✓
VIEW	✓	—	✓
ASSERTION	✓	—	✓
PROCEDURE	✓	✓	✓
FUNCTION	✓	✓	✓
TRIGGER	✓	—	✓
ROLE	✓	—	✓
PRIVILEG	✓	—	✓



2.2 Creating, Altering, and Removing a Domain

The syntax of the definition of a domain is as follows:¹

CREATE DOMAIN *<domain-name>* [**AS**] *<data-type>*
 [*<default-clause>*] [*<domain-constraint-list>*]

<default-clause> ::= **DEFAULT** *<default-value>*

<domain-constraint> ::= [*<constraint-name>*] *<check-constraint>*
 [*<characteristics>*]

¹ In the following grammars, **terminal** and *<non-terminal>* symbols are distinguished using different font types. Optional symbols are enclosed by [] brackets. The symbol | is used to list alternatives.

**<characteristics> ::= [[(NOT) DEFERRABLE]
INITIALLY {IMMEDIATE | DEFERRED }]**

A domain constraint is expressed by a check constraint which restricts the values of the specified data type to the permitted ones. The *default* clause is used to specify a default value for the domain.

The *characteristics* clause specifies the checking mode of a constraint. The checking mode determines the relative time when the constraint has to be checked within a transaction. In the *immediate* mode, the constraint is checked at the end of each database modification (either an insert, update or delete SQL-statement) that might violate the constraint. In the *deferred* mode, the checking is delayed until the end of the transaction.

In addition, the *characteristics* clause determines the initial checking mode, which must be valid for the constraint at the beginning of every transaction. Only *deferrable* constraints can be set to the *deferred* mode. The checking mode of a *non-deferrable* constraint always is *immediate*. The modes *initially immediate* and *non-deferrable* are implicit, if no other is explicitly specified. If *initially deferred* is specified, then *non-deferrable* shall not be specified, and thus *deferrable* is implicit.

The checking mode of a constraint can also be changed during the execution of a transaction using the following command:

**SET CONSTRAINTS {ALL | <constraint-name-list>}
{IMMEDIATE | DEFERRED }**

Example 1. Suppose in our application domain, three different cities are distinguished: 'Munich', 'London', and 'Paris'. The “default city” is 'Munich'. Such a domain can be created as follows:

**CREATE DOMAIN cities CHAR (6)
DEFAULT 'Munich'
CHECK(VALUE IN ('Munich', 'London', 'Paris'));**

□

The definition of a domain can be changed by setting/removing the default value or by adding/removing a constraint to/from the domain. The syntax of the alter domain statement is as follows:

ALTER DOMAIN <domain-name> <alter-domain-action>

**<alter-domain-action> ::= SET <default-clause>
| DROP DEFAULT
| ADD <domain-constraint>
| DROP CONSTRAINT <constraint-name>**

For each column that is based on the domain to be altered by removing the default value, the dropped default value is placed in that column if it does not already contain a default value. Analogously, for each column that is based on the domain to be altered by removing a domain constraint, the dropped domain constraint is attached to the constraint list of that column.

A domain can be dropped from the database schema using the following command:

DROP DOMAIN <domain-name> {**RESTRICT** | **CASCADE** }

If **RESTRICT** is specified, then the domain must not be referenced in any of the following: table column, body of an SQL routine, query expression of a view, or search condition of a constraint.

Let *c* be a column of a table *t* that is based on a domain *d*. If **CASCADE** is specified, then removing *d* results in the following modifications of *c*:

- The domain *d* is substituted by a copy of its data type.
- The default clause of *d* is included in *c*, if *c* does not contain an own default clause.
- The constraints of *d* are added to the table *t*.

2.3 Creating, Altering, and Removing a User-Defined Type

The main corpus of the syntax of the definition of a user-defined type is as follows:

```
CREATE TYPE <type-name> [UNDER <type-name>]
    [AS {<predefined-type> | <attribute-def-list>}]
    [{INSTANTIABLE | NOT INSTANTIABLE}]
    {FINAL | NOT FINAL}
    [<ref-type-spec>] [<method-spec-list>]
```

```
<attribute-def> ::= <attribute-name> <data-type>
    [<ref-scope-check>] [<default-clause>]
```

```
<ref-scope-check> ::= REFERENCES ARE [NOT] CHECKED
    ON DELETED <ref-action>
```

```
<ref-action> ::= NO ACTION
    | RESTRICT
    | CASCADE
    | SET NULL
    | SET DEFAULT
```

```
<ref-type-spec> ::= REF USING <predefined-type>
    | REF FROM ( <attribute-name-list> )
    | REF IS SYSTEM GENERATED
```

An attribute is a component of a structured type. A reference attribute is based on the reference type. The *reference scope check* clause shall only be specified for such reference attributes. Using this clause, one can specify whether and how to react on the deletion of a referenced instance. The *reference type specification* defines the way how the reference is created.

Every user-defined type is *instantiable* by default, that is, an instance of a user-defined type can be created unless it is explicitly disallowed by specifying the keyword **NOT INSTANTIABLE**.

The *under* clause is used to create a subtype of another structured type. In this way, type hierarchies can be built. The *under* clause shall not be used for distinct types since it is obviously not reasonable. Let **type2** be a subtype of **type1**, then **type2** inherits all attributes of **type1**. Here, **type2** shall not contain any attribute that has the same name as an inherited one. That is, attribute redefinition is not allowed.

The *final* clause indicates whether or not the structured type can be used as a supertype. Surprisingly, the keyword **NOT FINAL** must always be specified within the definition of a structured type. If the *under* clause is specified, the reference type specification is prohibited. For each attribute of a structured type *observer* and *mutator* methods are generated. These methods are used to access and modify the value of an attribute.

In case of the definition of a distinct type, the keyword **FINAL** must always be specified, while neither the *under* clause nor the reference type specification are allowed.²

Example 2. The following statement defines a distinct type:

```
CREATE TYPE swiss_francs AS DECIMAL (12,2) FINAL;
```

A structured type is defined as follows:

```
CREATE TYPE address AS (  
  street    VARCHAR(35),  
  number    DECIMAL(4),  
  zip        DECIMAL(5),  
  city       VARCHAR(25),  
  country    VARCHAR(30)  
) NOT FINAL;
```

The types defined above can now also be used within the definition of another structured type:

```
CREATE TYPE employee AS (  
  id          SMALLINT,  
  name        ROW(first VARCHAR(15), last VARCHAR(20)),  
  address      address,  
  supervisor    REF(employee) REFERENCES ARE CHECKED  
                                     ON DELETE SET NULL,  
  hiredate     DATE,  
  salary        swiss_francs  
) NOT FINAL;
```

In this case references are checked automatically whenever an instance of the referenced type is deleted. If the deletion concerns an actually referenced instance, then the attribute **supervisor** of the referencing instance is set to **NULL**.

We now define a subtype of the structured type above:

² Since the keywords **NOT FINAL** and **FINAL** must always be used without any options, it is not understandable why they have been introduced.

```
CREATE TYPE manager UNDER employee AS (
    bonus      swiss_francs
) NOT FINAL;
```

Managers are thus modeled as special employees having an additional bonus salary. □

An existing structured type can also be changed by adding new attributes or method specifications and by removing existing attributes or method specifications. The main corpus of the syntax of the alter type statement looks as follows:

```
ALTER TYPE <type-name> <alter-type-action>

<alter-type-action> ::= ADD ATTRIBUTE <attribute-def>
                        | DROP ATTRIBUTE <attribute-name> RESTRICT
                        | ADD <method-spec>
                        | DROP <routine> RESTRICT

<routine> ::= {PROCEDURE | FUNCTION} <routine-name>
```

The attribute or routine to be dropped shall not be contained in any of the following: body of an SQL routine, query expression of a view, search condition of a constraint or assertion, or trigger action.

A user-defined type is dropped using the following command:

```
DROP TYPE <type-name> {RESTRICT | CASCADE}
```

If **RESTRICT** is specified, then the user-defined type to be dropped, among others, shall not be referenced in any of the following: another user-defined type, expression of a view, search condition of a constraint or assertion, or trigger action.

2.4 Creating, Altering, and Removing a Table

As already mentioned, there are two types of tables: (1) usual tables as known from the previous SQL standard and (2) typed tables which are based on a structured type.

The main corpus of the syntax of a table definition is follows:

```
CREATE TABLE <table-name>
    {( <table-element-list> )
     | OF <type-name> [UNDER <table-name>]
     [( <table-element-list> ) ]}

<table-element> ::= <column-def>
                  | <table-constraint-def>
                  | REF IS <column-name> <ref-generation>
                  | <column-name> WITH OPTIONS <option-list>
```

```

<column-def> ::= <column-name> <type-or-domain-name>
                [<ref-scope-check>] [<default-clause>]
                [<column-constraint-def-list>]

<column-constraint-def> ::= [CONSTRAINT <constraint-name>]
                             <column-constraint> [<characteristics>]

<column-constraint> ::= NOT NULL
                       | UNIQUE
                       | PRIMARY KEY
                       | CHECK ( <search-condition>)
                       | <ref-spec>

<ref-spec> ::= REFERENCES <table-name> ( <column-name-list>)
               [MATCH {SIMPLE | PARTIAL | FULL } ]
               [ON UPDATE <ref-action>] [ON DELETE <ref-action>]

<table-constraint-def> ::= [CONSTRAINT <constraint-name>]
                             <table-constraint> [<characteristics>]

<table-constraint> ::= UNIQUE ( VALUE )
                     | UNIQUE ( <column-name-list>)
                     | PRIMARY KEY ( <column-name-list>)
                     | CHECK ( <search-condition>)
                     | FOREIGN KEY ( <column-name-list>) <ref-spec>

<ref-generation> ::= SYSTEM GENERATED
                     | USER GENERATED
                     | DERIVED

<option-list> ::= [<scope-clause>] [<default-clause>]
                 [<column-constraint-def-list>]

<scope-clause> ::= SCOPE <table-name>

```

A usual table is defined by specifying a column list, whereas a typed table is defined using the *of* clause with the name of a structured type. In the latter case, the attributes of the structured type determines the schema of the table. The column options are used to define default values and constraints for a typed table.

Using the *under* clause, table hierarchies can be built by setting typed tables into a subtable relationship. The typed table specified in the under clause refers to the direct supertable of the created typed table. Every typed table may have at most one direct supertable. Besides, a subtable must not have an explicit primary key.

Let *table1* be a table of type *type1* and *table2* a table of type *type2*. If *table1* occurs in the under clause of the definition of *table2*, then *type2* must be a direct subtype of *type1*.

Example 3. The following statement defines a typed table based on the structured type introduced in Example 2:

```
CREATE TABLE employees OF employee;
```

A subtable of this table is defined using the under clause, for instance, as follows:

```
CREATE TABLE managers OF manager UNDER employee;
```

Note this table has the same schema as the following usual table:

```
CREATE TABLE managers (  
  id          SMALLINT,  
  name        ROW(first VARCHAR(15), last VARCHAR(20)),  
  address     address,  
  supervisor  REF(employee) REFERENCES ARE CHECKED  
                                ON DELETE SET NULL,  
  hiredate    DATE,  
  salary      swiss_francs,  
  bonus       swiss_francs  
);
```

A main difference between these two kinds of *managers* tables is that the rows of the typed table can be referenced in the sense of object-orientation. That is, there may be a reference column referring to an instance of that typed table. In this case, the value of the reference column is a row (or object) identifier associated with a row of the typed table. In contrast, the only way to reference a row in a usual table is to use the foreign key concept. Here, the value of the (referencing) foreign key must match the value of a (referenced) unique/primary key of that table. □

The definition of a table can be changed using the alter table statement, which has the following syntax:

```
ALTER TABLE <table-name> <alter-table-action>
```

```
<alter-table-action> ::= ADD [COLUMN] <column-def>  
                        | ALTER [COLUMN] <column-name> <col-action>  
                        | DROP <column-name> {RESTRICT | CASCADE }  
                        | ADD <table-constraint-def>  
                        | DROP <constraint-name> {RESTRICT | CASCADE }  
  
<col-action> ::= SET <default-clause>  
                | DROP DEFAULT  
                | ADD <scope-clause>  
                | DROP SCOPE {RESTRICT | CASCADE }
```

The alter table statement can only be applied to base tables. A typed table, however, cannot be altered. Usual base table can be altered by adding and

removing columns and constraints. Besides, an existing column of such a table can be altered by setting/removing the default value. Furthermore, the scope of a reference column can be added or removed. A scope can only be added if the reference column does not already have one.

If **RESTRICT** is specified for the *drop column* clause, then the column to be dropped shall not be contained in any of the following: body of an SQL routine, query expression of a view, search condition or triggered action of a trigger, or search condition of a table constraint.

A primary key can only be added to a table that has no supertable.

If **RESTRICT** is specified for the *drop constraint* clause, then the following must hold: neither a table constraint nor a view shall be dependent on the table constraint to be dropped and its name shall not be contained in the body of any SQL routine body.

A table is dropped from the database using the following command:

```
DROP TABLE <table-name> {RESTRICT | CASCADE}
```

Removing a table means that the table schema as well as the table instance are removed together with the corresponding privileges.

If **RESTRICT** is specified, then the table to be dropped shall not have any subtable, and moreover it shall not be referenced in any of the following: body of an SQL routine, scope of the declared type of an SQL routine parameter, query expression of a view, search condition or triggered action of a trigger, search condition of a check constraint of another table, search condition of an assertion, or a referential constraint of another referenced table. If **CASCADE** is specified, such dependent schema elements are dropped implicitly.

2.5 Creating and Removing a View

SQL-99 supports two types of views: (1) usual views and (2) typed views that are based on a structured type.

The main corpus of the syntax of the view definition is as follows:

```
CREATE VIEW <table-name>
    {( <column-name-list> )
    | OF <type-name> [UNDER <table-name>]
    [( <column-option-list> ) ]}
AS <query-expression>
[WITH CHECK OPTION ]

<column-option> ::= <column-name> WITH OPTIONS <scope-clause>
```

A usual view is defined by a column list, whereas a typed view is specified using the *of* clause which determines the schema of the view. The column option list is used to specify the scope of reference columns.

The *under* clause is used to create a subview of another typed view. In this way, view hierarchies can be built. The typed view specified in the under clause

refers to the direct superview of the created typed view. Every typed view may have at most one direct supertable.

Let **view1** be a view of type **type1** and **view2** be a view of type **type2**. If **view1** occurs in the under clause of the definition of **view2**, then **type2** must be a direct subtype of **type1**.

The *check* option ensures that all data modification statements performed on the view will be validated against the query expression of that view.

Example 4. Assuming there is a structured type **employee** and a typed table **employees**, the following statement defines a typed view:

```
CREATE VIEW cheap_employees OF employee AS (  
  SELECT * FROM employees WHERE salary < 5000  
);
```

□

A view definition cannot be altered, but it can be dropped using the following statement:

```
DROP VIEW <table-name> {RESTRICT | CASCADE }
```

If **RESTRICT** is specified, then the view to be dropped shall neither have any subviews nor it shall be referenced in any of the following: body of an SQL routine, scope of the declared type an SQL routine parameter, query expression of another view, search condition or triggered action of a trigger, search condition of a check constraint of another table, search condition of an assertion, or a referential constraint of another referenced table. If **CASCADE** is specified, such dependent schema elements are dropped implicitly.

2.6 Creating and Removing an Assertion

An assertion is created using the following statement:

```
CREATE ASSERTION <assertion-name> CHECK ( <search-condition> )  
  [<characteristics>]
```

In contrast to the search condition of a column constraint or a table constraint, the search condition of an assertion may also refer to more than one row of one or more tables, that is, table-level and database-level check constraints can be defined within an assertion.

An existing assertion is dropped from the database using the following statement:

```
DROP ASSERTION <assertion-name>
```

2.7 Creating, Altering, and Removing a Routine

A routine in SQL-99 refers to a procedure or function. The main corpus of the syntax of a procedure and function definition is as follows:

```
CREATE PROCEDURE <routine-name> ( <parameter-list> )
                               <routine-characteristics> <routine-body>
```

```
CREATE FUNCTION <routine-name> ( <parameter-list> ) <returns-clause>
                  <routine-characteristics> <routine-body>
```

Loosely spoken, a function is a procedure with an additional *return* clause. A routine can be specified with different characteristics. For instance, a routine can be either an SQL or an external routine, it can be deterministic or non-deterministic, and it can be a routine that only reads or modifies SQL data. The routine body consists of an SQL procedure statement.

A routine can also be altered and dropped, respectively, using the following commands:

```
ALTER <routine> <alter-routine-characteristics> RESTRICT
```

```
DROP <routine-name> {RESTRICT | CASCADE}
```

If **RESTRICT** is specified, then the routine to be dropped shall not be referenced in any of the following: body of an SQL routine, query expression of a view, search condition of a check constraint or assertion, or triggered action of a trigger. If **CASCADE** is specified, such dependent schema elements are dropped implicitly.

2.8 Creating and Removing a Trigger

The syntax of a trigger definition is as follows:

```
CREATE TRIGGER <trigger-name>
{BEFORE | AFTER }
{INSERT | DELETE | UPDATE [OF <column-name-list>]}
ON <table-name> [REFERENCING <old-or-new-values-list>]
[FOR EACH {ROW | STATEMENT }]
[WHEN ( <search-condition> ) ]
<SQL-procedure-stat> | BEGIN ATOMIC <SQL-procedure-stat-list> END
```

```
<old-or-new-values> ::= {OLD | NEW } [ROW] [AS] <correlation-name>
                      | {OLD | NEW } TABLE [AS] <table-alias>
```

A trigger is implicitly activated when the specified event occurs. The activation times *before* and *after* specify when the trigger should be fired, that is, either before the triggering event is performed or after the triggering event. Valid triggering events are the execution of insert, update, or delete statements. A trigger condition and trigger action can be verified and executed, respectively, for each

row affected by the triggering statement or once for the whole triggering event (for each statement). Trigger conditions and actions can refer to both old and new values of the rows affected by the triggering event.

An existing trigger can be dropped using the following command:

```
DROP TRIGGER <trigger-name>
```

2.9 Creating and Removing Roles

The create role statement has the following syntax:

```
CREATE ROLE <role-name> [WITH ADMIN OPTION <grantor>]
```

After the creation of a role, no privileges are associated with that role. These have to be added using the grant statement, as described in the following. The *admin* option is used to give the grantee the right to grant the role to others, to revoke it from other users or roles, and to drop or alter the granted role.

An existing role is dropped using the following statement:

```
DROP ROLE <role-name>
```

2.10 Granting and Revoking Privileges

Privileges are granted to a user or role using the grant statement, which has the following syntax:

```
GRANT {ALL PRIVILEGES | <privileges-or-role-name-list>}  
  TO <grantee-list>  
  [WITH HIERARCHY OPTION] [WITH GRANT OPTION]  
  [WITH ADMIN OPTION] [GRANTED BY <grantor>]
```

The *hierarchy* option can only be applied to privileges on typed tables or typed views. It specifies that the granted privilege is also valid for all subtables (sub-views) of a typed table (typed view). The *grant* option is used to specify that the granted privilege is also grantable, that is, the user is allowed to give others the privilege to access and use the named object. In general, the hierarchy and grant options shall only be specified when privileges are granted while the admin option shall only be specified when roles are granted.

A granted privilege or role can be revoked from a user or role using the revoke command. The syntax of the revoke command is as follows:

```
REVOKE [{GRANT | HIERARCHY | ADMIN} OPTION FOR]  
  {ALL PRIVILEGES | <privileges-or-role-name-list>}  
  FROM <grantee-list> [GRANTED BY <grantor>]  
  {RESTRICT | CASCADE}
```

Analogously to the grant statement, the hierarchy and grant option shall only be specified when privileges are revoked, while the admin option can only be specified when roles are revoked.

Example 5. The following statement creates a role `reademp`. This role is associated with the privilege to read the data of all kinds of employees:

```
CREATE ROLE reademp;  
GRANT SELECT ON employee TO reademp WITH HIERARCHY OPTION;
```

The hierarchy option ensures that all users associated with the role `reademp` are also allowed to read the data of any special employee, for instance, the salary of a manager.

It is also possible to revoke only the hierarchy option from the role `reademp`. This is achieved by executing the following statement:

```
REVOKE HIERARCHY OPTION FOR SELECT ON employee FROM reademp;
```

Using the statement above without the hierarchy option revokes the privilege to select *any* employee. □

3 Comparison of Schema Evolution Constructs in SQL-99 and Commercial DBMS

In this section, we compare the schema evolution language constructs of SQL-99 [Int99] with that of the commercially available (object-)relational database management systems Oracle8i Server (Release 8.1.6) [Ora99], IBM DB2 Universal Database (Version 7) [IBM00], Informix Dynamic Server.2000 (Version 9.2) [Inf99], Microsoft SQL Server (Version 7.0) [Mic99], Sybase Adaptive Server (Version 11.5) [Syb99], and Ingres II (Release 2.0) [Ing99]. In the following, we will use the abbreviations Oracle, DB2, Informix, MSSQL, Sybase, and Ingres, respectively, to refer to these systems. In addition, we will use the term *reference systems* to refer to all of these systems as a whole. It should be pointed out that in fact only Oracle, DB2, and Informix could be denoted as object-relational database management systems. The other three systems are pure relational database management systems.

3.1 Domains and Assertions

Neither the concept of a domain nor the concept of an assertion is supported by any reference system.

However, there are a few rudimentary approaches in that directions. Ingres, for instance, provides the concept of an integrity rule which actually corresponds to a row-level assertion. Internally, these integrity rules are stored with a generated integer number, which is used to identify an integrity rule within a table definition. This number is needed, for instance, to drop an integrity rule. An integrity rule is created and dropped, respectively, as follows:

```
CREATE INTEGRITY ON <table-name> IS <search-condition>  
DROP INTEGRITY ON <table-name> {ALL | <integer-list>}
```

The creation of an integrity rule fails if the table contains a row that does not satisfy the search condition. In the Ingres manuals, there is a hint to define check constraints within a create table or alter table statement instead of specifying integrity rules that anyway are not conform to the standard.

MSSQL and Sybase provide language constructs for specifying named default values and rules. A named default value is created and dropped, respectively, as follows:

```
CREATE DEFAULT <default-name> AS <constant-expression>  
DROP DEFAULT <default-name>
```

A named default value can then be bound to a table column or a distinct type using the predefined stored procedure **SP_BINDEFAULT** with the following parameters:

```
SP_BINDEFAULT <default-name>, '<column-or-type-name>'
```

Before a named default value can be dropped, it must be unbound from all dependent schema elements using the predefined stored procedure **SP_UNBINDEFAULT**.

In the context of MSSQL and Sybase, a rule defines a domain of acceptable values for a particular table column or distinct type. A rule is created and dropped, respectively, as follows:

```
CREATE RULE <rule-name> AS <search-condition>  
DROP RULE <rule-name>
```

A rule must be unbound using the predefined stored procedure **SP_UNBINDRULE** before it can be dropped.

Similarly to a named default value, a rule is bound to a table column or distinct type using the predefined stored procedure **SP_BINDRULE** with the following parameters:

```
SP_BINDRULE <rule-name>, '<column-or-type-name>'
```

When a rule is bound to a table column or distinct type, it specifies the acceptable values that can be inserted into that column. A rule, however, does not apply to data that already exists in the database at the time the rule is created. It also does not override a column definition. That is, a nullable column can take the null value even though **NULL** is not included in the text of the rule. If both a default and a rule are defined, the default value must fall in the domain defined by the rule. A default value that conflicts with a rule will never be inserted. An error message will be generated each time such a conflicting default value is tried to be inserted. Since a rule performs some of the same functions as a check constraint, the latter, standard way of restricting the values in a column is recommended.

3.2 User-Defined Types □

As already mentioned, SQL-99 supports two kinds of user-defined types: distinct type and structured types. Since certain user-defined types have been provided by some of the reference systems prior to the introduction of SQL99, the notions and language constructs used in these systems differ in some cases.

Table 2 gives an overview of the support of the various named and unnamed type constructors in the reference systems. Interestingly, the unnamed array type is supported in none of the reference systems, although it is proposed in SQL-99. On the other hand, the unnamed collection types set, multiset, and list are only provided in Informix. These types were originally included in the preliminary drafts of SQL-99, but they were now postponed to the next version of the standard, which is currently referred to as SQL4.

Table 2. Comparison of User-Defined Types

TYPE		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
Named	DISTINCT	✓	—	✓	✓	✓	✓	—
	OBJECT (Structured)	✓	✓	✓	—	—	—	—
	ROW	—	—	—	✓	—	—	—
	VARRAY	—	✓	—	—	—	—	—
	TABLE	—	✓	—	—	—	—	—
Unnamed	ROW	✓	—	—	✓	—	—	—
	SET	—	—	—	✓	—	—	—
	MULTISET	—	—	—	✓	—	—	—
	LIST	—	—	—	✓	—	—	—
	ARRAY	✓	—	—	—	—	—	—
	REF	✓	✓	✓	—	—	—	—
HIERARCHY (UNDER)		✓	—	✓	✓	—	—	—

In DB2 and Informix, the creation of a distinct type is performed using the following command:

```
CREATE DISTINCT TYPE <type-name> AS <source-type-name>
```

In DB2, the statement above must end with an additional keyword **WITH COMPARISONS** unless the source data type is **BLOB**, **CLOB**, **LONG VARCHAR**, **LONG VARGRAPHIC**, or **DATALINK**. This option ensures that the instances of the same distinct type can be compared. Both, DB2 and Informix automatically generate two functions to cast in both directions from the distinct type to its source type and vice versa.

DB2's syntax of the definition of a structured type is more or less the same as proposed in SQL-99. Nevertheless, it is worth to mention that DB2 requires

the specification of an awkward, non-optional keyword **MODE DB2SQL**. On the other hand, the specification of the keyword **NOT FINAL** is optional.

DB2 also provides a means to alter the definition of a structured type. Attributes and methods can be added (dropped) to (from) a structured type. The syntax of the alter type statement is as follows:

```
ALTER TYPE <type-name>
    { ADD ATTRIBUTE <attribute-def>
      | DROP ATTRIBUTE <attribute-name> [RESTRICT]
      | ADD <method-spec>
      | DROP METHOD <method-name> [RESTRICT] }
```

The *restrict* option ensures that no attribute or method can be dropped if the structured type they belong to is referenced in any other schema element.

Distinct and structured types are dropped in DB2 using the following commands:

```
DROP DISTINCT TYPE <type-name>
DROP TYPE <type-name>
```

A user-defined type is not dropped if there is any schema element that depends on this type. An error occurs if the user-defined type to be dropped has a subtype or is used within the definition of a column, typed table, typed view, or another structured type.

Informix supports the concept of a structured type under the notion of a *named row type*. The syntax of the definition of a named row type is as follows:

```
CREATE ROW TYPE <type-name> ( <attribute-def-list> )
    [UNDER <type-name>]
```

A named row type can be used to create a typed table or typed view. It can also be assigned to a column of a table or to an attribute of another named row type. The concept of subtyping is supported analogously to SQL-99. That is, attributes and methods are inherited from the supertypes to the subtypes and the redefinition of inherited attributes and methods is not allowed. In a type hierarchy, a named row type cannot be substituted for its supertype or its subtype.

An attribute of a named row type can be defined as non-nullable. Other kinds of constraints, however, cannot be applied to a named row type directly. They have to be defined within the create table or alter table statement.

Besides these two kinds of user-defined types, Informix also supports the unnamed *row* type as well as the collection types *set*, *multiset*, and *list*. Complex data types are created by combining these type constructors in any order.

Distinct types and named row types are dropped in Informix using the following commands:

```
DROP TYPE <type-name> RESTRICT
DROP ROW TYPE <type-name> RESTRICT
```

Since the keyword **RESTRICT** is mandatory, a user-defined type cannot be dropped if the database contains any schema element that depends on this type.

Oracle distinguishes three kinds of user-defined types: *object types*, *varying array types*, and *table types*, which are defined according to the following syntax:

```
CREATE TYPE <type-name> AS OBJECT ( <attr-method-spec> )
CREATE TYPE <type-name> AS VARRAY OF <data-type>
CREATE TYPE <type-name> AS TABLE OF <data-type>
```

Oracle does not support the concept of subtyping. The notion of an object type corresponds to the notion of a structured type in SQL-99. Distinct types are not supported. Instead, two named collection types are provided. A varying array type defines an ordered multiset of elements, each of which has the same data type. The data type of the elements have to be one of the following: built-in data type, reference type, or object type. The cardinality of the multiset must be explicitly specified. A table type in fact defines an unordered multiset of elements, each of which has the same data type. The elements can be either instances of an object type or values of a built-in type. The cardinality of this multiset is not restricted. A collection type, however, cannot contain any other collection type. That is, a varying array type, for instance, cannot contain any elements that are varying arrays or tables. In this way, nesting of tables is restricted to one level.

Nevertheless, Oracle allows to alter a type definition, either by recompiling a type definition or by replacing the object type. The syntax of the alter type statement looks as follows:

```
ALTER TYPE <type-name> AS
    { COMPILE | REPLACE AS OBJECT ( <attr-method-spec> ) }
```

A user-defined type is dropped in Oracle using the following command:

```
DROP TYPE <type-name> [FORCE ]
```

This statement removes a user-defined type if there is no schema element in the database that relies on this type. If there is such a dependent schema element, the *force* option can be used to drop the type and to mark all columns that use this type as *unused*.

The concept of a distinct type is supported in MSSQL and Sybase, too. In these systems, a distinct type is created and dropped, respectively, by executing the following predefined stored procedures:

```
SP_ADDTYPE <type-name>, '<predefined-type>'
SP_DROPTYPE <type-name>
```

A distinct type cannot be dropped if it is referenced in any other schema element.

Table 3 summarizes the various schema evolution operations related to user-defined types.

Table 3. Comparison of Type Constructs

		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
Distinct Type	CREATE	✓	—	✓	✓	(✓)	(✓)	—
	ALTER	—	—	—	—	—	—	—
	DROP	—	—	✓	—	(✓)	(✓)	—
	RESTRICT	✓	—	—	✓	—	—	—
	CASCADE	✓	—	—	—	—	—	—
Structured Type (Object/Row)	CREATE	✓	✓	✓	✓	—	—	—
	UNDER	✓	—	✓	✓	—	—	—
	ALTER	✓	(✓)	✓	—	—	—	—
	DROP	—	✓	✓	—	—	—	—
	RESTRICT	✓	—	—	✓	—	—	—
	CASCADE	✓	(✓)	—	—	—	—	—

3.3 Tables

In all reference systems, the relational part of a table definition basically follows the proposal of SQL-99. In the following, we therefore focus more on the object-relational extensions of a table definition.

While typed tables are supported in Oracle, DB2, and Informix, table hierarchies (subtables) are only provided in DB2 and Informix. The main corpus of the definition of typed tables in all three systems more or less follows the definition of SQL-99. Here, the systems mainly differ in the naming of the same concepts. DB2 uses the terms of SQL-99 (one should better say that SQL-99 uses the terms of DB2), Informix also uses the term typed table but the notion of a named row type instead of the term structured type, and Oracle calls these concepts object tables and object types.

In all three systems, user-defined types can be used as data type of a column. As we could see in the previous subsection, various type constructors are provided in the different systems to create complex data types.

Tables can be altered in all reference systems. However, the provided alter table constructs differ in several ways. Table 4 gives an overview of the different constructs.

As we can see there, all reference systems provide means to add new columns and constraints to a table. Sybase, however, has the restriction that the newly added column must be nullable. In all reference systems, it is also possible to drop an existing constraint from a table. A column, however, can only be dropped in Oracle, Informix, MSSQL, and Ingres.

Some of the reference systems distinguishes between different *drop* options. Ingres is the only reference system that follows the proposal of SQL-99 with respect to the removing of a column or constraint. The specification of the drop option is mandatory and it can be decided between the options **RESTRICT** and

Table 4. Comparison of **ALTER TABLE** Constructs

ALTER TABLE <table-name>		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
ADD COLUMN		✓	✓	✓	✓	✓	✓	✓
ALTER COLUMN	SET <data-type>	—	(✓)	(✓)	(✓)	(✓)	—	—
	SET DEFAULT	✓	(✓)	—	(✓)	—	(✓)	—
	DROP DEFAULT	✓	—	—	—	—	—	—
	CONSTRAINT	—	(✓)	—	(✓)	(✓)	—	—
	ADD SCOPE	✓	—	✓	—	—	—	—
	DROP SCOPE	✓	—	—	—	—	—	—
DROP COLUMN		—	✓	—	✓	✓	—	—
	RESTRICT	✓	—	—	—	—	—	✓
	CASCADE	✓	—	—	—	—	—	✓
ADD CONSTRAINT		✓	✓	✓	✓	✓	✓	✓
DROP CONSTRAINT		—	✓	✓	✓	✓	✓	—
	RESTRICT	✓	—	—	—	—	—	✓
	CASCADE	✓	✓	—	—	—	—	✓
ADD TYPE		—	—	—	✓	—	—	—

CASCADE. The former requires that the drop column or drop constraint statement is rejected if there is a schema element that depends on the schema element to be dropped. The latter implicitly drops the dependent schema elements. Oracle supports the cascade option in combination with the drop constraint construct. If **CASCADE** is not explicitly specified, the default mode implements the restrict semantics. The other reference systems do not support any drop options. The default mode of the drop constraint statement is **RESTRICT** in MSSQL and Sybase, while it is **CASCADE** in DB2 and Informix. A drop column construct (without a drop option) is also provided in Oracle, Informix, and MSSQL. The default mode is **RESTRICT** in Oracle and MSSQL, while it is **CASCADE** in Informix.

All reference systems except Ingres provide an alter column construct. In Table 4 we used the symbol ‘(✓)’ to mark the constructs that do not follow the syntax of the SQL-99 proposal. The syntax and semantics of the alter column constructs provided in the different systems differ in several ways:

Oracle: **ALTER TABLE** <table-name> **MODIFY** <column-name>
[<data-type>] [<default-clause>] [**NOT NULL**]

DB2: **ALTER TABLE** <table-name> **ALTER COLUMN** <column-name>
{**SET DATA TYPE** <data-type>
| **ADD SCOPE** <typed-table-name>}

Informix: **ALTER TABLE** <table-name> **MODIFY** <column-name>
<data-type> [<default-clause>] [<constraint>]

```
MSSQL: ALTER TABLE <table-name> ALTER COLUMN <column-name>
      <data-type> [NOT NULL | NULL ]

Sybase: ALTER TABLE <table-name> REPLACE <column-name>
      <default-clause>
```

That is, Oracle and Informix provides a construct to alter the data type, default value, and constraints of a column. In fact, Oracle only allows the specification of a not null constraint; other constraints have to be added or dropped using the add constraint and drop constraint statements, as discussed previously. In MSSQL, a column can be altered by changing its data type and defining a not null constraint for this column. DB2 provides a construct that can be used either to change the data type of a column or to add a scope to a reference column. Sybase can replace the default value of a column.

Informix allows to convert a usual table into a typed one. This modification is performed using the *add type* clause. The added type must be compatible with the impicit type of the usual table, that is, they must have exactly the same attributes with respect to their names and data types.

Although all reference systems provide a drop table statement, they implement this statement with different options and semantics (for an overview see Table 5).

Table 5. Comparison of **DROP TABLE** Constructs

		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
DROP TABLE		—	✓	✓	✓	✓	✓	✓
	RESTRICT	✓	—	—	✓	—	—	—
	CASCADE	✓	✓	(✓)	✓	—	—	—

DB2 applies the *cascade or invalidate* semantics, meaning that the content of the table is removed together with all dependent indexes, constraints, and privileges, while dependent views, procedures, functions, and triggers are only invalidated. If a table contains a subtable, it cannot be dropped before all its subtables are dropped. DB2 provides the *hierarchy* option to drop all tables of a table hierarchy. Note that the functionality of the option is included in the *cascade* option of SQL-99.

In MSSQL and Sybase, a drop table statement removes the table definition together with all data, indexes, triggers, constraints, and privileges for that table. Any view, stored procedures, default, or rule that references the dropped table must be dropped explicitly.

Oracle applies the restricts semantics to drop a table. Nevertheless, it supports the specification of the cascade option to drop a table together with all

dependent constraints. As in DB2, dependent views, procedure, functions, and triggers are not dropped. They are only invalidated and can later be used if the table is re-created.

Informix supports the restrict semantics as well as the cascade semantics. If neither **RESTRICT** nor **CASCADE** is specified, the drop table statement is executed with the cascade semantics.

Ingres drops a table implicitly with the cascade semantics, although it does not support the explicit specification of this option. □

3.4 Views

The three object-relational systems Oracle, DB2, and Informix support both usual (untyped) views as well as typed views. However, subviews (view hierarchies) are only provided in DB2. Table 6 gives an overview of various schema evolution operations defined on the concept of a view.

Table 6. Comparison of **VIEW** Constructs

		SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
CREATE	VIEW ...	✓	✓	✓	✓	✓	✓	✓
	VIEW ... OF ...	✓	✓	✓	✓	—	—	—
	VIEW ... OF ... UNDER	✓	—	✓	—	—	—	—
ALTER VIEW ...		—	✓	✓	—	✓	—	—
DROP	VIEW ...	—	✓	✓	✓	✓	✓	✓
	VIEW ... RESTRICT	✓	—	—	✓	—	—	—
	VIEW ... CASCADE	✓	—	(✓)	✓	—	—	—

Let us now first consider the main corpus of the syntax of the view definition in Oracle:

```

CREATE VIEW <table-name>
  [( <column-name-list>) | OF <type-name>]
  AS <query-expression>
  [WITH {CHECK OPTION | READ ONLY }]
  
```

A typed view is defined using the *of* clause, as in SQL-99. Oracle provides the *read-only* option which ensures that no insert, update, or delete can be performed through the view on the underlying base table(s). The well-known *check* option validates inserts, updates, and deletes against the query expression of the view and rejects invalid changes through the view.

The view definition in Informix is very similar to the previous one, except that the read only option is not supported and the keyword **OF TYPE** must be used instead of the keyword **OF** to define a typed view:

```
CREATE VIEW <table-name>
    [( <column-name-list> ) | OF TYPE <type-name>]
AS <query-expression> [WITH CHECK OPTION ]
```

Compared to the previous view definitions, DB2 provides a more advance one, which allows to define view hierarchies based on typed views. For the definition of a subview, DB2' create view statement, however, requires the awkward, non-optional keywords **MODE DB2SQL** and **INHERIT SELECT PRIVILEGES**. The main part of the create view statement looks as follows:

```
CREATE VIEW <table-name>
    [( <column-name-list> ) | OF <type-name>
    [MODE DB2SQL
    UNDER <table-name>
    INHERIT SELECT PRIVILEGES ]]
AS <query-expression> [WITH CHECK OPTION ]
```

Oracle, DB2, and MSSQL provide an alter view statement. However, since the alter view statement is not standardized yet, the different implementations provide different functionality under the same label. In Oracle, the alter view statement only recompiles a view:³

```
ALTER VIEW <table-name> COMPILE
```

In DB2, the alter view statement modifies an existing view by altering a reference column to add a scope. The syntax of this statement is as follows:

```
ALTER VIEW <table-name> ALTER [COLUMN] <column-name>
ADD SCOPE <typed-table-name>
```

Finally, in MSSQL, the alter view statement replaces a previously created view without affecting dependent stored procedures or triggers and without changing privileges. The syntax of this variant is as follows:

```
ALTER VIEW <table-name> [( <column-name-list> ) ]
AS <query-expression> [WITH CHECK OPTION ]
```

Concerning the drop view statement, all six reference systems more or less closely follows the proposal of SQL-99. In Oracle, DB2, MSSQL, and Sybase, the execution of a drop view statement invalidates all views that are based on the view to be dropped. DB2 additionally provides the *hierarchy* option, which is similar to the *cascade* option in SQL-99. The hierarchy option is used to implicitly drop all views of a view hierarchy. The syntax of the corresponding statement is as follows:

```
DROP VIEW HIERARCHY <table-name>
```

Here, table name refers to the name of a root view.

³ A view can be replaced in Oracle using the keyword **CREATE OR REPLACE VIEW** in create view command.

In Informix, a view can be dropped following either the restrict or the cascade semantics. **RESTRICT** ensures that the drop view operation fails if any existing view is defined on the view to be dropped. **CASCADE** guarantees that all such dependent view are implicitly dropped, too. If none of these keywords is explicitly specified, the drop operation is executed with the cascade semantics. Ingres also applies this strategy, but without providing any options. The remaining reference systems apply the restrict semantics.

3.5 Procedures, Functions, and Triggers

All reference systems support the creation and deletion of routines and triggers. However, since the programming languages to define these routines and triggers differ in several ways, we omit a comparison of the various programming styles. Instead, we address some other interesting issues.

As the alter view statement, the alter routine and alter trigger statements are not standardized yet. Nevertheless, they are included in some of the reference systems. For instance, the alter procedure statement is used in Oracle to recompile a (stand-alone) procedure:

Oracle: **ALTER {PROCEDURE | FUNCTION | TRIGGER} <routine-name>
COMPILE**

MSSQL allows to alter an existing procedure without changing privileges and without affecting any dependent stored procedures or triggers. Analogously, MSSQL provides an alter trigger statement that replaces the definition of an existing trigger.

Oracle, Informix, and MSSQL even support the enabling and disabling of triggers:

Oracle: **ALTER TRIGGER <trigger-name> {ENABLE | DISABLE}
ALTER TABLE <table-name> {ENABLE | DISABLE} ALL TRIGGERS**
Informix: **SET TRIGGERS <trigger-name-list> {ENABLED | DISABLED}**
MSSQL: **ALTER TABLE <table-name> {ENABLE | DISABLE} TRIGGER
{ALL | <trigger-name-list>}**

In Oracle, DB2, MSSQL, and Sybase, the execution of a drop routine statement invalidates all schema elements that are based on the routine to be dropped. In Informix and Ingres, a procedure is dropped implicitly with the cascade semantics.

3.6 Roles and Privileges

Roles and privileges are supported by all six reference systems in a very similar way as proposed in SQL-99. Table 7 gives an overview of the corresponding language constructs.

As depicted there, the concept of a role is provided in all reference systems except DB2. Concerning the creation of a role, Oracle, Informix, and Ingres

Table 7. Comparison of **ROLE**, **GRANT**, and **REVOKE** Constructs

			SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
CREATE ROLE			✓	✓	—	✓	(✓)	(✓)	✓
DROP ROLE			✓	✓	—	✓	(✓)	(✓)	✓
ALTER ROLE			✓	✓	—	✓	(✓)	(✓)	✓
GRANT			✓	✓	✓	✓	✓	✓	✓
	WITH GRANT OPTION		✓	✓	✓	✓	✓	✓	✓
REVOKE			—	✓	✓	✓	✓	✓	—
		RESTRICT	✓	—	—	✓	—	—	✓
		CASCADE	✓	✓	—	✓	—	—	✓
	GRANT OPTION FOR		—	—	✓	—	✓	✓	—
		RESTRICT	✓	—	—	—	—	—	✓
		CASCADE	✓	—	—	—	✓	✓	✓

closely follows the SQL-99 proposal. MSSQL and Sybase, in contrast, implement the concept of a role by providing predefined stored procedures. In MSSQL, a role is created by executing the stored procedure **SP_ADDAPPROLE** with the following parameters:

SP_ADDAPPROLE <role-name>, <password>

After creation a role is inactive by default. It can be activated by executing the stored procedure **SP_SETAPPROLE** with the same parameters as above. A role is dropped by executing the stored procedure **SP_DROPAPPROLE** with the name of the role:

SP_DROPAPPROLE <role-name>

In Sybase, a role is granted and revoked, respectively, by executing the stored procedure **SP_ROLE** as follows:

SP_ROLE {'GRANT' | 'REVOKE'}, <predefined-role> <user-name>

Sybase supports three predefined roles:

- 1. **SA_ROLE** (system administrator),
- 2. **SSO_ROLE** (system security officer), and
- 3. **OPER_ROLE** (operator).

A role is switched on or off, respectively, using **SET ROLE {ON | OFF}**.

With respect to the grant statement, all reference systems follow the SQL-99 proposal. Even the *grant* option is provided by all systems. In case of the revoke statement, however, there are some minor differences in the various implementations.

In Sybase, the revoke statement is implemented with the cascade semantics, that is, the removal of a privilege implies the removal of all dependent privileges. The cascade semantics is also default in Informix. Oracle, Informix, and Ingres provide the keyword **CASCADE** to explicitly specify this semantics. The restricted semantics prevents from revoking a privilege if there is a dependent privilege.

Applying the revoke statement with the *grant option* revokes the right to grant the granted privilege to others. If additionally the *cascade* option is used, the transitively granted privileges are revoked, too. This option is supported in MSSQL, Sybase, and Ingres.

3.7 Constraints

Although constraints are part of a table definition, we discuss their evolution separately and in more detail due to their importance. As mentioned before, new constraints can be added to a table and existing ones removed from a table. In addition, and in contrast to SQL-99, the checking of constraints can be enabled and disabled in some of the reference systems. These issues can even be combined. For instance, a constraint can be added to a table in the disabled mode or a disabled constraint can be enabled but without verifying it against the current content of the corresponding table. Table 8 gives an overview of the support of constraint evolution constructs in SQL-99 and in the reference systems.

Table 8. Comparison of Constraint Evolution Constructs

			SQL99	Oracle	DB2	Informix	MSSQL	Sybase	Ingres
ADD	ENABLE	VALIDATE	(✓)	✓	(✓)	✓	✓	(✓)	(✓)
		VALIDATE + EXCEPTION	—	✓	—	✓	—	—	—
		NOVALIDATE	—	✓	—	—	✓	—	—
	DISABLE		—	✓	—	✓	—	—	—
DROP			—	✓	✓	✓	✓	✓	—
		RESTRICT	✓	—	—	—	—	—	✓
		CASCADE	✓	✓	—	—	—	—	✓
ENABLE		VALIDATE	—	✓	—	✓	—	—	—
		VALIDATE + EXCEPTION	—	✓	—	✓	—	—	—
		NOVALIDATE	—	✓	—	—	✓	—	—
DISABLE			—	✓	—	✓	✓	—	—
		RESTRICT	—	—	—	—	—	—	—
		CASCADE	—	✓	—	—	—	—	—

According to Table 8, Oracle supports the full range of schema evolution constructs that are related to constraints. Another interesting fact is that SQL-99 does not provide any means to enable and disable constraints. The gray shaded fields highlight the default settings. The ‘(✓)’ marked fields state that the concept is supported implicitly. For instance, the *add* and *drop* clauses are provided by all reference systems in its standard form. However, extensions like **ENABLE**, **DISABLE**, or **CASCADE** are not supported explicitly by the ‘(✓)’ marked systems. In the following we will discuss the different constructs in more detail.

When a constraint is added to a table using the statement

ALTER TABLE <table-name> **ADD** <table-constraint>

the newly added constraint is enabled and validated by default. *Enabled* means that future modifications of the content of the table will be verified against this constraint (unless it disabled in the meanwhile). *Validated* means that the content of table is verified against the constraint when the latter is added to the table.

All reference systems support these two modes. Moreover, Oracle and Informix allow to add an enabled constraint even in case there is a row in the table that does not satisfy the constraint. In this case an *exception* clause has to be specified as follows:

Oracle: **ALTER TABLE** <table-name> **ADD** <table-constraint>
EXCEPTIONS INTO <table-name>

Informix: **ALTER TABLE** <table-name> **ADD** <table-constraint> **FILTERING**

The rows that do not satisfy the newly added constraint are removed from the table to an exception/diagnostic table, which can be named explicitly in Oracle. Both systems also allow to add a disabled constraint to a table. By default, such a constraint is not validated when it is added to the table. A disabled constraint is specified as follows:

Oracle: **ALTER TABLE** <table-name> **ADD** <table-constraint> **DISABLE**

Informix: **ALTER TABLE** <table-name> **ADD** <table-constraint> **DISABLED**

Oracle and MSSQL allow to add an enabled constraint that is not validated when it is added to the table. In this case, some rows in the table may violate the constraint. However, future modifications of the table will be verified against the newly added constraint. Such a constraint is defined as follows:

Oracle: **ALTER TABLE** <table-name> **ADD** <table-constraint>
NOVALIDATE

MSSQL: **ALTER TABLE** <table-name> **WITH NOCHECK**
ADD <table-constraint>

In all reference systems, a constraint can be dropped from a table as follows:

ALTER TABLE <table-name> **DROP** <constraint-name>

Some of the reference systems support the specification of *drop* option, which is either **RESTRICT** or **CASCADE**. **RESTRICT** disallows the removal of the constraint if there is another constraint that depends on the constraint to be dropped. If **CASCADE** is specified, the constraint is dropped together with all its depending constraints.

The specification of one of these modes is mandatory in Ingres, whereas it is optional in Oracle. The other systems do not support the specification of these modes. The default mode is **RESTRICT** in Oracle, Informix, MSSQL, and Sybase, and it is **CASCADE** in DB2.

A disabled constraint can be enabled in the validate mode as follows:⁴

Oracle: **ALTER TABLE** *<table-name>* **ENABLE** *<constraint>*

Informix: **SET CONSTRAINTS** *<constraint-name>* **ENABLED**

The enabling of a constraint can also be performed in the exception/filtering mode. All rows that violate the enabled constraint are removed from the table into an exception/violations table.

The enabling of a disabled constraint in the novalidate mode is specified as follows:

Oracle: **ALTER TABLE** *<table-name>* **ENABLE NOVALIDATE** *<constraint>*

MSSQL: **ALTER TABLE** *<table-name>* **CHECK** *<constraint>*

An enabled constraint can be disabled as follows:

Oracle: **ALTER TABLE** *<table-name>* **DISABLE** *<constraint>*

Informix: **SET CONSTRAINTS** *<constraint-name>* **DISABLED**

MSSQL: **ALTER TABLE** *<table-name>* **NOCHECK** *<constraint>*

The statements are executed in all three systems with the restrict semantics. Cascaded disabling of constraints is only supported by Oracle. For that, the keyword **CASCADE** has to be attached to the *disable* clause.

3.8 Renaming Schema Elements

In the following, we present a useful schema evolution operation that is already implemented in some of the reference systems, although it is not included in SQL-99.

The renaming of a table is supported by Oracle, DB2, Informix, MSSQL, and Sybase. In Oracle, DB2, and Informix, the syntax of the rename statement is as follows:

RENAME TABLE *<old-table-name>* **TO** *<new-table-name>*

⁴ Here, *<constraint>* stands for one of the following: **CONSTRAINT** *<constraint-name>*, **PRIMARY KEY**, or **UNIQUE** (*<column-list>*).

Oracle additionally provides the following alternative way to change the name of a table:

ALTER TABLE *<old-table-name>* **RENAME TO** *<new-table-name>*

Informix even allows to rename a particular column of a table applying the rename command with the following syntax:

RENAME COLUMN *<table-name>*. *<old-col-name>* **TO** *<new-col-name>*

MSSQL and Sybase provide a predefined stored procedure which is executed with the following parameters to rename a schema element:

SP_RENAME *<old-name>*, *<new-name>*

This procedure is applicable to names that refer to tables, columns, defaults, constraints, rules, triggers, views, and distinct types.

Note that renaming a schema element may also effect dependent schema elements. Oracle, for instance, automatically transfers the new name of a table to all dependent constraints, indexes, and privileges, while it invalidates the dependent views, functions, procedures, and triggers. DB2 applies a more strict strategy. The renaming of a table is disallowed if the table contains a check or referential constraint or there is a dependent view, trigger, function, procedure, or another table with a dependent constraint or reference column. If there is no such a dependency, the renaming is performed by updating the schema catalog and transferring the new name to all dependent indexes and privileges. In Informix, in contrast, the renaming is completely transparent, that is, the new name is transferred to the schema catalog as well as to all dependent schema elements.

4 Some Final Remarks

In this paper, we presented and compared the way schema evolution is supported in SQL-99 and in commercially leading (object-)relational database management systems. We will close this paper with a few remarks on some open issues and schema evolution operations that are available neither in SQL-99 nor in one of the reference systems.

An important open issue concerns the consistency of a schema after performing a schema evolution operation. This issue includes the question whether or not a schema definition as a whole is syntactically and semantically (logically) correct. Considering the current implementations of commercial database management systems, we can state that all systems perform syntactic checking. They, for instance, check whether a foreign key definition is correct in the sense that the names and the data types of the referencing and the referenced columns match. However, none of the systems perform (advanced) semantic checking. Suppose there is a table on which the check constraint **CHECK** ($y > 0$) is defined. Unfortunately, all reference systems accept an alter table statement that adds a new



(obviously contradicting) constraint of the form **CHECK** ($y < 0$). In fact, the reference systems do not provide any support for detecting inconsistent specifications implied by check constraints. Interestingly, efficient consistency checking procedures for important and often used kinds of constraints are provided, for instance, in [Ull89,SKN89,GSW96a,GSW96b]. Since the knowledge about the consistency problem and its solutions is highly important for a good design and correct evolution of a database, database designers and administrators have been aware of this problem. In object-relational database systems, the problem of inconsistent constraints becomes even more prominent because constraints are implicitly defined for all subtables of a table. In other words, there are constraints that are valid for a table on which they originally were not defined. So it becomes much harder to design, implement, and maintain a semantically correct database (schema).

Now turn the focus on schema evolution in object-oriented databases. Considering the research in this field, for instance, [BKKK87,Ngu89,TK90,SZ90,Bra93] [ABDS94,ST94,RR95,FMZ⁺95,Bel96,PÖ97], some nice schema evolution operations could be exploited for object-relational databases. For instance, a prominent schema evolution operation in an object-oriented database is the restructuring of a type or table hierarchy. Existing types or tables can be linked via a subtype or subtable, respectively. Such schema evolution operations are not supported by any current object-relational system. One could think about including statements of the forms

```
SET <subtype-name> UNDER <supertype-name>
SET <subtable-name> UNDER <supertable-name>
```

or

```
ALTER TYPE <subtype-name> ADD UNDER <supertype-name>
ALTER TABLE <subtable-name> ADD UNDER <supertable-name>
```

into the standard as well as commercial systems. Such statements would help to easily set existing types (tables) into a subtype (subtable) relationship. The inverse statements to drop a subtype or subtable relationship could look as follows:

```
ALTER TYPE <subtype-name> DROP UNDER <supertype-name>
ALTER TABLE <subtable-name> DROP UNDER <supertable-name>
```

One might also think about altering a subtype or subtable relationship by redirecting the link to another subtype or subtable, respectively. Another useful schema evolution construct could be the removal of a subtable from a table hierarchy without removing all its subtables. Instead these subtables could be directly linked to the supertable of the dropped table. An inline transformation [SCR01] can be used to flatten a column that is based on a structured type. Such a structured column is substituted by a set of columns which originally were the fields of that structured column.

The list of potentially useful schema evolution operations could be supplemented easily. Therefore, we close this paper by expressing our hope that the

next versions of the SQL standard and in particular of the commercial database systems will provide some more advanced schema evolution language constructs, which are hopefully embedded in a more clear and rigorously developed object-relational model.

Acknowledgments. Thanks to Kerstin Schwarz for useful remarks on a preliminary version of this paper.

References

- [ABDS94] E. Amiel, M.-J. Bellosta, E. Dujardin, and E. Simon. Supporting Exceptions to Behavioral Schema Consistency to Ease Schema Evolution in OODBMS. In J. B. Bocca, Matthias Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Data Bases, VLDB'94, Santiago, Chile, September 12–15, 1994*, pages 108–119. Morgan Kaufmann Publishers, San Francisco, CA, 1994.
- [Bel96] Z. Bellahsene. A View Mechanism for Schema Evolution in Object-Oriented DBMS. In R. Morrison and J. B. Keane, editors, *Advances in Databases: 14th British National Conf. on Databases, BNCOD 14, Edinburgh, UK, July 1996*, Lecture Notes in Computer Science, Vol. 1094, pages 18–34. Springer-Verlag, Berlin, 1996.
- [BKKK87] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In U. Dayal and I. Traiger, editors, *Proc. of the 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, CA*, ACM SIGMOD Record, Vol. 16, No. 3, pages 311–322, ACM Press, 1987.
- [Bra93] S. E. Bratsberg. *Evolution and Integration of Classes in Object-Oriented Databases*. Dissertation, The Norwegian Institute of Technology, University of Trondheim, June 1993.
- [FMZ⁺95] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O₂ Object Database System. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proc. of the 21st Int. Conf. on Very Large Data Bases, VLDB'95, Zürich, Switzerland, September 11–15, 1995*, pages 170–182. Morgan Kaufmann Publishers, San Francisco, CA, 1995.
- [GSW96a] S. Guo, W. Sun, and M. A. Weiss. On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):604–616, August 1996.
- [GSW96b] S. Guo, W. Sun, and M. A. Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM Transactions on Database Systems*, 21(2):270–293, June 1996.
- [IBM00] IBM Corporation. *IBM DB2 Universal Database: SQL Reference, Version 7*, 2000.
- [Inf99] Informix Corporation, Menlo Park, CA. *Informix Guide to SQL: Syntax, Informix Dynamic Server.2000, Version 9.2*, December 1999.
- [Ing99] Ingres Corporation. *Ingres Database Administrator's Guide, Version II*, 1999.

- [Int99] International Organization for Standardization (ISO) & American National Standards Institute (ANSI), ANSI/ISO/IEC 9075-2:99. *ISO International Standard: Database Language SQL - Part 2: Foundation (SQL/Foundation)*, September 1999.
- [Mic99] Microsoft Corporation. *Microsoft SQL Server, Version 7.0*, 1999.
- [Ngu89] G. T. Nguyen. Schema Evolution in Object-Oriented Database Systems. *Data & Knowledge Engineering*, 4(1):43–67, July 1989.
- [Ora99] Oracle Corporation. *Oracle8i SQL Reference, Release 8.1.6*, December 1999.
- [PÖ97] R. J. Peters and M. T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transactions on Database Systems*, 22(1):75–114, March 1997.
- [RR95] Y.-G. Ra and E. A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. In P. S. Yu and A. L. P. Chen, editors, *Proc. of the 11th IEEE Int. Conf. on Data Engineering, ICDE'95*, pages 165–172. IEEE Computer Society Press, Los Alamitos, CA, 1995.
- [SCR01] H. Su, K. T. Claypool, and E. A. Rundensteiner. Extending the Object Query Language for Transparent Metadata Access. In H. Balsters, B. De Brock, and S. Conrad, editors, *Database Schema Evolution and Meta-Modeling: 9th International Workshop on Foundations of Models and Languages for Data and Objects (FOMLADO/DEMM 2000)*, Lecture Notes in Computer Science Vol. 2065, pages 181–200, Springer-Verlag, 2001.
- [SKN89] X. Sun, N. N. Kamel, and L. M. Ni. Processing Implications on Queries. *IEEE Transactions on Software Engineering*, 15(10):1168–1175, 1989.
- [ST94] M. H. Scholl and M. Tresch. Evolution towards, in, and beyond Object Databases. In K. von Luck and H. Marburger, editors, *Management and Processing of Complex Data Structures, Proc. of the 3rd Workshop on Information Systems and Artificial Intelligence, Hamburg, Germany, February/March 1994*, Lecture Notes in Computer Science, Vol. 777, pages 64–82. Springer-Verlag, Berlin, 1994.
- [Syb99] Sybase Inc. *Transact-SQL User Guide, Version 11.0*, 1999.
- [SZ90] A. Skarra and S. B. Zdonik. Type Evolution in an Object-Oriented Database. In A. F. Cárdenas and D. McLeod, editors, *Research Foundations in Object-Oriented and Semantic Database Systems*, chapter 6, pages 137–155, Series in Data and Knowledge Base Systems, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [TK90] L. Tan and T. Katayama. Meta Operations for Type Management in Object-Oriented Databases: A Lazy Mechanism for Schema Evolution. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases, Proc. of the 1st Int. Conf., DOOD'89, Kyoto, Japan, December, 1989*, pages 241–258. North-Holland, Amsterdam, 1990.
- [Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, Rockville, MD, 1989.

Consistency Management in Runtime Evolving Concurrent Information Systems: A CO-NETS-Based Approach

Nasreddine Aoumeur* and Gunter Saake

Otto-von-Guericke-Universität Magdeburg, Institut für Technische und Betriebliche
Informationssysteme, Postfach 4120, D-39016 Magdeburg, Germany,
{aoumeur,saake}@iti.cs.uni-magdeburg.de

Abstract. For adequately specifying and rapid-prototyping concurrent information systems, we proposed in [AS99] a new form of object oriented (OO) Petri nets. Referred to as CO-NETS, this approach allows in particular to conceive such systems as complex autonomous yet cooperating components. Moreover, for coping with intrinsic dynamic evolution in such systems, we have straightforwardly extended this proposal by introducing notions of meta-places, non-instantiated transitions and a two-step evaluated inference rule [Aou00].

The purpose of this paper is to tackle with another crucial dimension characterizing real-world information systems, namely static and dynamic integrity constraints. For this aim, we propose to associate with each component a ‘constraints’ class. To enforce such constraints, we propose an appropriate ‘synchronization’ inference rule that semantically relates ‘constraints’ transitions with intrinsically dependent ones in the associated component. For a more flexible consistency management we enrich this first proposal by an adequate meta-level, where constraints may be dynamically created, modified or deleted. Finally, we show how this proposal covers a large number of constraint subclasses, including life-cycle based constraints and constraints based on complex derived information as view classes.

1 Introduction

The high growing rate of nowadays organizations requires more and more complex information systems for their support. This complexity is expressed by several requirements which have to be fulfilled by these systems. Moreover, it is nowadays commonly recognized that apart from efficiency and user-friendly features of an intended information system, all its other (functional and non-functional) aspects have to be *rigorously* addressed in the decisive phase of *specification / validation* (and verification). Among the requirements that are considered as the milestones of any advanced information system, recent research advocates particularly the following[PS98]:

* This work is supported by a DAAD grant.

Reactivity and componentization: Information systems are software conceived to be running for a very long time with a continuous communication with their environment, that is, they are reactive systems in the sense of [MP92]. On the other hand, today's information systems are more and more regarded as very complex, loosely connected and multi-layered components.

Full distribution and communication: Due to the ubiquity of standardized distributed architectures, information system components are mostly distributed over different (geographical) sites, behaving in a true concurrent way, and requiring in most cases different forms of (synchronous / asynchronous) communication.

Dynamic evolution[SCT01]: Another crucial dimension that characterizes complex information systems is the frequent change of most of their functionalities (and architecture) over the time. Such a change is particularly triggered by new market laws, international economies change, user's need change, etc.

Complex integrity constraints: In addition to the above characteristics, information systems supporting real-world organization tasks should also reflect policies and procedures governing such an organization. In other words, in an information system we should be able to express and respect all constraints related to the universe of discourse of the application at hand.

Heterogeneity[SL90]: Due to different sources and forms of data and knowledge in complex information systems, the heterogeneity dimension also represents one of the most challenging features. Indeed, this dimension has led to a completely independent area known as federated information (and databases) systems.

Following the object oriented paradigm—as the best existing paradigm around it most of the above information system facets may be addressed—we proposed in [AS99] an appropriate integration of object-oriented structuring mechanisms into a variety of algebraic Petri nets. On the basis of several non-trivial case studies [AS00b][AS00c], we showed the appropriateness of this integration—we referred to as CO-NETS approach—for coping with the two above first requirements in a satisfactory way. Indeed, first the CO-NETS approach allows to construct complex components as a hierarchy of classes with explicit interfaces—using different forms of inheritance and aggregations. Second, CO-NETS components behave in a true concurrency way by exhibiting intra- as and inter-object concurrency as well as different forms of (synchronous and asynchronous) communication. Third, while CO-NETS components autonomously evolve, they may interact with each other using their explicit interfaces. Fourth, transitions governing the behaviour in such components are interpreted in rewriting logic, which allows for deriving rapid-prototypes using concurrent rewriting techniques.

Besides that, to cope with the above third characteristic we have extended this proposal for adequately handling runtime modification of component behaviour [Aou00]. This extension is based on the following. First, in each component behaviour we distinguish between a 'fixed forever' part reflecting minimal properties of the application at hand and a possibly subject to future modifica-

tion part. Second, we construct a meta-level composed of a meta-place containing instead of object states a complete behaviour captured as tokens and of three transitions for creating, modifying or deleting a given behaviour while the system is still running. Finally, we connect the two levels using syntactically appropriate read-arcs and semantically a two-step evaluated inference rule.

The purpose of this paper is to tackle with the consistency management through an adequate specification as well as enforcement of complex static and dynamic integrity constraints. More precisely, the main ideas for consistency management of CO-NETS components, we propose in this paper, may be highlighted in the following.

- For each conceived CO-NETS component, we construct an intrinsic class we call a constraints class. Object states in such a class are tuples recording necessary information associated with a given integrity constraint. With each tuple, one or more ‘constraint’ transitions are associated; they reflect the allowed change in such tuple attributes. Finally, for enforcing the respect of such constraints we relate them to method transitions (in the component) which may affect them.
- In order to allow a runtime introduction, modification or deletion of any constraint, we enrich this first **proposal** with some reflection capabilities by building a meta-level. This meta-level is mostly inspired by our already proposal for handling dynamic modification of component behaviour, but with several specificities.
- On the basis of this two-level approach for consistency management, we demonstrate its applicability to several subclasses of constraints carefully studied in [DB00]. These subclasses include life-cycle based constraints, constraints based on computing complex derived information as view classes, and last but not least constraints involving more than one component. □

The remaining sections of this paper are as follows. Section 2 reviews some CO-NETS aspects and its extension for handling runtime behaviour evolution. In the main section we present the specification and enforcement of integrity constraints as we highlighted above. In the conclusion, we sketch the achieved work and outline some further steps for a more consistency using the CO-NETS approach.

2 CO-NETS with Runtime Evolution: **An Overview**

In this section we recall some features of the CO-NETS approach that are relevant to the purpose of this paper. In this sense using a simple bank account example, we review the main ideas in specifying complex information systems using this approach. Moreover, to deal with runtime modification in such systems, we revisit the extension proposed in [Aou00] by mainly introducing a more adequate inference rule.

2.1 CO-NETS: Specification of Simple Components

Our main ideas in specifying complex information systems by integrating object-oriented concepts and high level Petri nets include: (1) an appropriate algebraic signature for templates; (2) a rigorous construction of object nets associated with such templates; and (3) a true concurrent interpretation of the behaviour of such nets using rewriting logic.

Template Signature. A template signature defines the structure of object states and the form of operations to be accepted by such states. In our approach, the template signature we propose can informally be described as follows.

- Object states are terms of the form

$$\langle Id|atr_1 : val_1, \dots, atr_k : val_k, at_bs_1 : val'_1, \dots, at_bs_{k'} : val'_{k'} \rangle$$

With:

- Id is an observed object identity taking its values from an appropriate abstract data type we denote by OId .
- atr_1, \dots, atr_k are considered to be local (i.e. hidden from the outside) attribute identifiers having as current values respectively val_1, \dots, val_k .
- As observed attributes (by other components) in an object state we consider $at_bs_1, \dots, at_bs_{k'}$; their associated current values are $val'_1, \dots, val'_{k'}$;
- A simple deduction rule, called ‘object-state splitting / merging’ rule, which permits to split (resp. to recombine) any object state when necessary is proposed. This deduction rule can be described as follows: $\langle Id|attrs_1, attrs_2 \rangle = \langle Id|attrs_1 \rangle \oplus \langle Id|attrs_2 \rangle$; with $attr_i$ as an abbreviation of a list $atr_{i1} : val_{i1}, \dots, atr_{ik} : val_{ik}$, and \oplus a multiset operator to be explained later.
- A clear distinction is made between local and external as imported / exported messages. Local messages allow changing object states in a given component, whereas external ones allow interacting different components using their observed attributes.

Example 1. We present a very simplified **Account** description. Each account is characterized by: its identifier as a concatenation of a natural number with the bank name, its balance of sort money, a minimal limit of its balance, and by the interest percent. As possible operations on such accounts we consider : the withdraw and deposit of a given amount as well as the increase of the interest percent. This account signature takes the following form.

obj Account is .

```

protecting money nat string interest .
sort Id.Account Account .
sort OPEN-AC WITHDRW DEPOSIT INTRS .
subsort Id.Account < OId .
(* the Account object state declaration *)
op ⟨_|Bal : _, Lmt : _, Ints : _⟩ : Id.Account money money
interest → Account .
```

```

(* Messages declaration *)
op OpenAc : Id.Account Id.Bank nat string → OPEN-AC .
op Wdr : Id.Account money → WITHDWR .
op Dep : Id.Account money → DEPOSIT .
op IncI : Id.Account interest → INTRS .
vars H : Id.Customer .
vars C : Id.Account .
vars W, D, L : money .
vars I, NI : Interest .
endo.

```

Please be aware that all data types like **nat**, **money**, and **string** are assumed to be algebraically specified elsewhere; also because we only consider this template we assume that all messages and attributes are local. We also note that all the specified variables will be used in the corresponding nets later. ■

Template specification. Given a template signature denoted by TS which captures the structural aspects of an information system, its behaviour is constructed by associating a CO-NET with this signature—leading to the notion of template specification that we denote by $SP = \langle TS, Net \rangle$. Informally speaking, the net associated with a given template signature is constructed as follows.

- The places of the net are precisely defined by associating with each message generator one ‘message’ place. Also, with each object state sort an ‘object’ place is associated. We denote the set of all places by P .
- Transitions, which may include conditions, reflect the effect of messages on object states (i.e. method body).

Example 2. By applying these translation ideas to the account signature we obtain the CO-NET depicted in Figure 1. In this net, the four message places correspond to the four message declarations, while the object place allows to capture the **Account** object instances. Four transitions reflecting the behaviour of these messages are conceived. It is worth mentioning that in each transition, the input as well as the output arcs are inscribed just by the *relevant part* of the invoked object state(s). For instance, in the DEP(osit) transition only the attribute (Bal)ance is invoked (i.e. $\langle C|Bal : B \rangle$ for the input arc and $\langle C|Bal : B + D \rangle$ for the output arc). This constitutes the key ideas for a full exhibition of the intra- (and inter-) object concurrency. As an example, the increase of interest method (i.e. the transition INTR) and the deposit method (i.e. the transition DEP) may be performed in parallel for a same account by appropriately splitting its state. ■

CO-NETS: Semantical Aspects. Given a CO-NET associated with a template specification, its semantics should provide us with permissible states that a marked CO-NET may be in. On the other hand, it should allow us formally deducing in a true-concurrency way any reachable state from an initial one. By permissible state we mainly understand the respect of the uniqueness of object identities and of the encapsulation property during object states change.

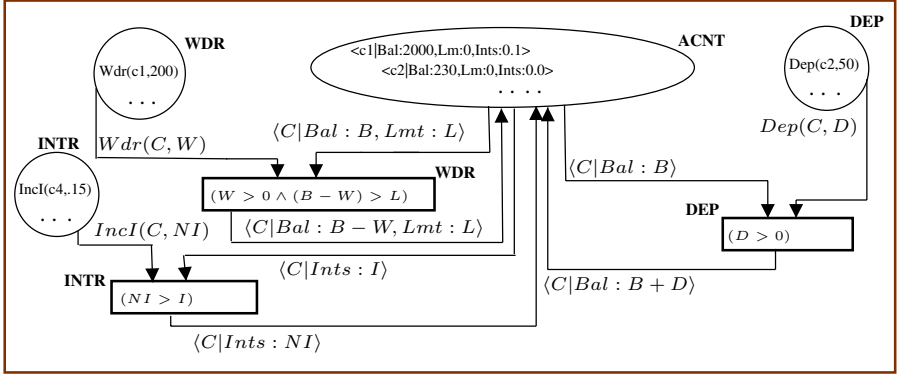


Fig. 1. The CO-NETS Account Specification

Objects creation and deletion. Regarding a marked CO-NET as a society of objects and messages imply that each object has to be uniquely identified with a persistent identity. In order to ensure this uniqueness and to allow the dynamic creation / deletion of objects, we propose the following conceptualization.

1. With each marked CO-NET modeling a component denoted as C_p a new place of sort $Id.obj$ ($< OId$) is associated. Such a place contains the *current object identifiers* of objects in C_p . □
2. For the creation of new objects, we introduce a new message sort (and a corresponding place) we denote by Ad_{C_p} . Also, we introduce a message generator for creating object states, we denote by ad_{C_p} and indexed by $Id.obj \times Ad_{C_p}$.
3. Each object state creation should be performed using the net depicted in the left hand side of Figure 2. The intended semantics for the notation \sim is that for firing the transition **NEW** the identifier Id should not already be in the place $Id.obj$ (i.e. the notation \sim captures the notion of an inhibitor arc). After firing this transition, there is an insertion of the new identifier Id in the place $Id.obj$ and creation of a new object namely $\langle Id | atr_1 : in_1, \dots, atr_k : in_k \rangle$; where in_1, \dots, in_k are optional initial attribute values.

Object state change. For evolving object states in a given component, we propose an appropriate general pattern for ‘local’ transitions. As depicted in Figure 3, this change pattern can be intuitively explained as follows. The contact of the only relevant parts—possible due to the state splitting/merging deduction rule—of some states, namely $\langle Id_1 | attrs_1 \rangle, \dots, \langle Id_k | attrs_k \rangle$ with some messages, namely $ms_{i_1}, \dots, ms_{i_p}$, declared in this component, and under conditions on the invoked attributes and message parameters, results in the following effect:

- the messages $ms_{i_1}, \dots, ms_{i_p}$ disappear;
- the states of some (parts of) objects participating in the communication change, namely I_{s_1}, \dots, I_{s_t} . Such change is symbolized by $attrs'_{s_1}, \dots, attrs'_{s_t}$

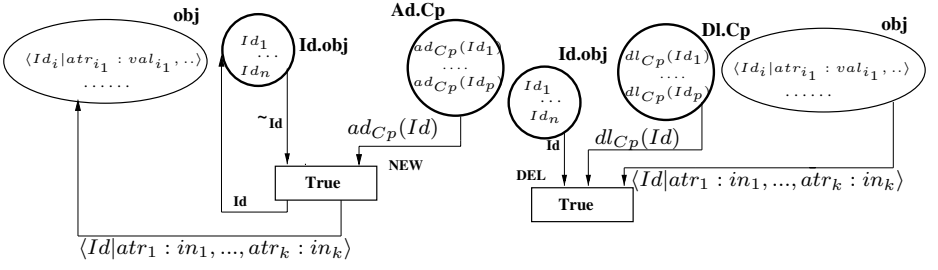


Fig. 2. Objects creation and deletion using OB-NETS

- instead of $attrs_{s_1}, \dots, attrs_{s_t}$. The other (unchanged parts of) object states are denoted by $attrs_{i_1}, \dots, attrs_{i_r}$ so that $\{i_1, \dots, i_r\} \cup \{s_1, \dots, s_t\} = \{1, \dots, k\}^1$;
- new messages (local or exported) are sent to objects of this component C_p , namely $ms_{h_1}, \dots, ms_{h_r}$ which may include (explicit) deletion and/or creation messages.

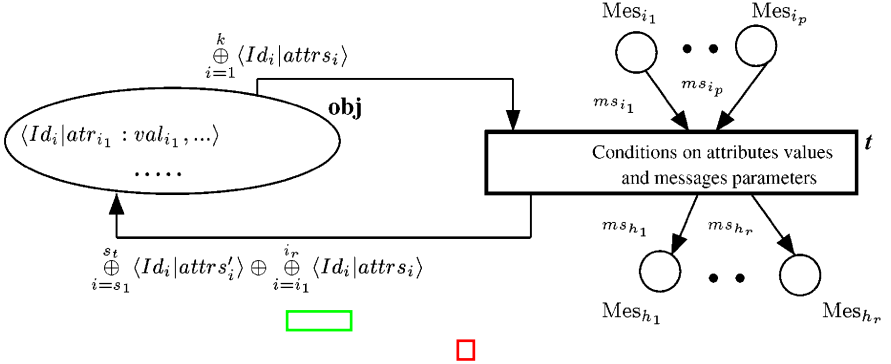


Fig. 3. A general intra-component evolution pattern

Following our approach for generating rewrite rules governing a given transition (see [AS99]), the corresponding rewrite rule for this general form of transitions (depicted in Figure 3) is:

$$\begin{aligned}
 \mathbf{t}: & \left(obj, \bigoplus_{i=1}^k \langle Id_i | attrs_i \rangle \right) \xrightarrow[p]{k=1} (Mes_{i_k}, ms_{i_k}) \Rightarrow \left(obj, \bigoplus_{k=1}^t \langle Id_{s_k} | attrs'_{s_k} \rangle \bigoplus_{k=1}^r \langle Id_{i_k} | attrs_{i_k} \rangle \right) \\
 & \bigotimes_{k=1}^r (Mes_{h_k}, ms_{h_k}) \text{ if } Condition \wedge M(Ad.Cp) = \emptyset \wedge M(Dl.Cp) = \emptyset.
 \end{aligned}$$

¹ In other words, there is no *implicit* creation or deletion of (parts) of object states—that would lead to inconsistency w.r.t. the described creation/deletion process in Figure 2.

Remark 1. The operator \otimes is defined as a multiset union and allows relating different place identifiers with their current markings. Moreover, we assume that \otimes is distributive over \oplus (i.e. $(p, mt_1 \oplus mt_2) = (p, mt_1) \otimes (p, mt_2)$ with mt_1, mt_2 as multisets of terms over \oplus and p a place identifier). The condition $(M(Ad.Cp) = \emptyset \wedge M(Dl.Cp) = \emptyset)$ ensures that any deletion and creation message should be performed at first. This allows avoiding any form of inconsistency like the manipulation of an object already logically deleted but physically still existing (i.e. there is a sending message for deleting it but not yet performed). \blacklozenge

Example 3. By applying this general form of rewrite rule to the account example, we result in the following rewrite rules:

WDR: $(WDR, Wdr(C, W)) \otimes (ACNT, \langle C|Bal : B, Lmt : L \rangle)$
 $\Rightarrow (ACNT, \langle C|Bal : B - W, Lmt : L \rangle)$ if $(W > 0) \wedge (B - W) > L$

DEP: $(DEP, Dep(C, D)) \otimes (ACNT, \langle C|Bal : B \rangle)$
 $\Rightarrow (ACNT, \langle C|Bal : B + D \rangle)$ if $(D > 0)$

INTR: $(INTR, IncI(C, NI)) \otimes (ACNT, \langle C|Ints : I \rangle)$
 $\Rightarrow (ACNT, \langle C|Int : NI \rangle)$ if $(NI > I)$

2.2 The Runtime Behaviour Modification in CO-NETS

In this subsection, first, we review the main ideas and corresponding constructions for handling runtime modification we proposed in [Aou00]. Then, we propose a more adequate inference rule for propagating a given behaviour from the meta-level to the object level.

Meta-place and non-instantiated transition constructions. For handling runtime modification in a given CO-NETS component, the constructions we proposed in [Aou00] may be summarized as follows.

1. In order to free some CO-NETS transitions from their rigidity, we propose to replace each of their three components—namely input tokens inscribing their input arcs, output tokens inscribing their output arcs and their conditions—by appropriate variables with same sorts respectively. We refer to such transitions with only variable inscriptions as *non-instantiated* transitions. Their general form is sketched in the lower right hand-side of Figure 4. In this general pattern for non-instantiated transitions, all (arc-) inscriptions—namely $IC_{obj}, IC_{i_1}, \dots, IC_{i_p}$ for input arcs, $CT_{obj}, CT_{h_1}, \dots, CT_{h_r}$ for output arcs and TC for conditions—are to be considered as variables.
2. We gather all (input and output) arc inscriptions as well as conditions, we have substituted by variable inscriptions, into a single *tuple*:

$\langle \text{tr_id: version} \mid (\text{input-})\text{multiset}, (\text{output-})\text{multiset}, \text{cond} \rangle$

In this tuple, while **tr_id** refer to the transition identifier the **version** reflects the possibility of associating more than a behaviour to a given transition. In particular, with respect to the general pattern of transitions in Figure 3, such

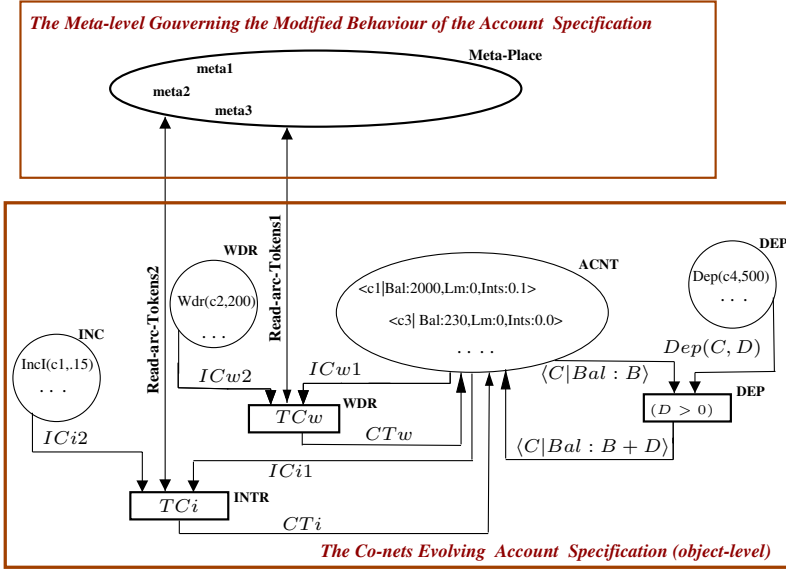
Table 1. Abbreviations in Figure 4

metal	$\langle T_k : n_1 (obj, \bigoplus_s \langle Id_1 attr_{s_i} \rangle) \otimes_r (Mes_{i_r}, mes_{i_r}), (obj, \bigoplus_h \langle Id'_1 attr'_{i_h} \rangle) \otimes_l (Mes_{h_l}, mes_{h_l}), TC_1 \rangle$
Add-meta	$Add_Bh(T, \bigotimes_i (P_i, IC_i), \bigotimes_j (Q_j, CT_j), TC)$
exist-meta	$\langle T : k IC, CT, TC \rangle$
notexist-meta	$\sim \langle T : k IC, CT, TC \rangle$
new-version	$\langle T : k + 1 \bigotimes_i (P_i, IC_i), \bigotimes_j (Q_j, CT_j), TC_j \rangle$
new-behaviour	$\langle T : 1 \bigotimes_i (P_i, IC_i), \bigotimes_j (Q_j, CT_j), TC_j \rangle$
cond1	$True$
cond2	$True$
del	$Del_Bh(T, i)$
dl-object	$\langle T : i -, -, - \rangle$
modif	$Chg_Bh(T, i, \bigotimes_j (P'_j, IC'_j), \bigotimes_h (Q'_h, CT'_h), TC')$
md-object	$\langle T : i \bigotimes_j (P'_j, IC'_j), \bigotimes_h (Q'_h, CT'_h), TC' \rangle$
to-md-object	$\langle T : i \bigotimes_i (P_i, IC_i), \bigotimes_r (Q_r, CT_r), TC \rangle$
selected-meta-Token	$\langle T : i (obj, IC_{obj}) \bigotimes_{i=i_1}^{i_p} (Mes_{i_i}, IC_{i_i}), (obj, CT_{obj}) \bigotimes_{j=h_1}^{h_r} (Mes_j, CT_j), TC \rangle$
objects	$\bigoplus_{i=1}^k \langle Id_i attr_{s_i} \rangle$
mdobjects	$\bigoplus_{i=s_1}^{s_t} \langle Id_i attr'_{s'_i} \rangle \oplus \bigoplus_{i=i_1}^{i_r} \langle Id_i attr_{s_i} \rangle$

□

methods may dynamically change. In other words their corresponding transitions should now be non-instantiated ones. Thus, we have to report their corresponding behaviour (firstly specified in Figure 1) as tokens in the place **Meta-place**. Moreover to illustrate such evolution, using the transition **ADD2** in the meta-level, we have added a new version for the withdraw method—captured by the token **meta3** in Figure 5. In this new version, in addition to the withdrawn amount a constant denoted *tax* has also to be subtracted from the balance in each withdraw operation, and the withdrawn amount should not exceed a certain percent of the balance (here 2 percent). In the same way the increase of interest method is modified by requiring the new percent should not exceed 0.01, and it is performed only if the balance is greater than 3000—this is captured by the modified token **meta2**. ■

The semantical counterpart as a meta-inference rule. For capturing theoretical interpretation of these constructions, we propose with respect to the same CO-NETS rewriting logic-based semantics an adequate inference rule. This inference rule can be regarded as a more flexible formulation of the one proposed in [Aou00]. The main ideas under the proposed reformulation are the following. First, we generate a rewrite rule associated with each non-instantiated transition in the same way as a usual CO-NETS transition except a new binary



Corresponding Abbreviations :

$$\begin{aligned}
 &\text{Meta-place tokens} \left\{ \begin{aligned}
 &\text{meta1 : } \langle WDR : 1 | (ACNT, \langle C | Bal : B, Lmt : L \rangle) \otimes (WDR, Wdr(C, W)), \\
 &\quad (ACNT, \langle C | Bal : B - W, Lmt : L \rangle), (W > 0 \wedge (B - W) > L) \rangle \\
 &\text{meta2 : } \langle INTR : 1 | (ACNT, \langle C | Ints : I, Bal : B \rangle) \otimes (INTR, IncI(C, NI)), \\
 &\quad (ACNT, \langle C | Ints : NI, bal : B \rangle), (NI > I) \wedge (NI < 0.01) \wedge (B > 3000) \rangle \\
 &\text{meta3 : } \langle WDR : 2 | (ACNT, \langle C | Bal : B, Lmt : L \rangle) \otimes (WDR, Wdr(C, W)), \\
 &\quad (ACNT, \langle C | Bal : B - tax - W, Lmt : L \rangle), \\
 &\quad (W > 0 \wedge ((B - W) > L) \wedge (W < .02 * B)) \rangle
 \end{aligned} \right. \\
 &\text{Read-arc} \left\{ \begin{aligned}
 &\text{Read-arc-Tokens1 : } hWDR : i | (ACNT, ICw1) \otimes (WDR, ICw2), (ACNT, CTw), TCw \\
 &\text{Read-arc-Tokens2 : } \langle INTR : i | (ACNT, ICi1) \otimes (INTR, ICi2), (ACNT, CTi), TCi \rangle
 \end{aligned} \right.
 \end{aligned}$$

Fig. 5. Runtime modification of the Account example using the CO-NETS extension

operator denoted $\|_r$ is proposed for separating read-arc inscriptions from the other place-tokens pairs. This operator is necessary because we should distinguish between tokens from the meta-level and those from the object level. This rule is considered as a non-instantiated rewrite rule, denoted t^{nins} , because it cannot be applied directly. That is way from this non-instantiated transition we derive a usual transition by selecting one behaviour as a token from the meta-place. This is achieved by applying different substitutions to corresponding variables in the read-arc token as explicitly described in the inference rule below. In this inference rule $M(P_{meta})$ represents the current marking of the place **meta-place**, while the notation $\| [T_{s(p_i)}] \|_{\oplus}$ represents a class of (multiset) terms (modulo the associativity, commutativity of \oplus) whose sort is exactly the one of the place p_i .

For each (meta-)rewrite rule :

$$\begin{aligned}
T^{ins} : (P_{meta}, \langle T : i \mid & \llbracket \bigotimes_{i=1}^k (p_i, IC_i) \rrbracket, \llbracket \bigotimes_{j=1}^l (q_j, CT_i) \rrbracket, TC_i \rangle \parallel_r \llbracket \bigotimes_{i=1}^k (p_i, IC_i) \rrbracket \\
\Rightarrow & \llbracket \bigotimes_{j=1}^l (q_j, CT_i) \rrbracket \text{ if } TC_i \text{ we have:} \\
& \exists \sigma_i \in [T_{s(p_i)}]_{\oplus}, \dots, \exists \sigma_j \in [T_{s(q_j)}]_{\oplus}, \exists \sigma \in [T_{bool}] \wedge \\
& \langle T : k \mid \llbracket \bigotimes_{i=1}^k (p_i, \sigma_i(IC_i)) \rrbracket, \llbracket \bigotimes_{j=1}^l (q_j, \sigma_j(CT_i)) \rrbracket, \sigma(TC_i) \rangle \in M(P_{meta}) \\
\hline
T^{ins} : & \llbracket \bigotimes_{i=1}^k (p_i, \sigma_i(IC_i)) \rrbracket \Rightarrow \llbracket \bigotimes_{j=1}^l (q_j, \sigma_j(CT_i)) \rrbracket \quad \text{if } \sigma(TC_i)
\end{aligned}$$

3 Consistency Management in CO-NETS

Let us first recall that ‘local’ constraints are already handled by our model. In fact, in the (pre-)condition associated with any transition we can introduce constraints which limit the application of such transitions just to objects and messages verifying some desired requirements. This can be easily extended for dealing with post-conditions by adding conditions acting on the appropriate resulting changes. Also in creating objects particular conditions may be set on their initial attribute values. Besides that and because attribute values as well as message parameters take their values from an (user-defined ordered sorted) algebraic specification, it is quite possible to associate more constraints on these entities, using in particular sort constraint primitives [GWM⁺92].

However, what go beyond the so-far introduced CO-NETS is, first, constraints acting on a *collection* of objects in a given component—like *cumulative* constraints [RSSS91]. In fact, all (pre- or post-) conditions that can be handled act just on the invoked objects and messages in a given method (i.e transition). Second, constraints involving more than one (global state of) component cannot be expressed. Third, constraints involving particular object life-cycles [DB00] also transcend our pre- and post- conditions locality. It is also very desirable to allow *runtime* modification (as well as creation and deletion) of integrity constraints, as we achieved it for component specification. It is worth mentioning however that we will not consider constraints involving more than two states (i.e. the current and previous states).

The purpose of the following three subsections is to introduce the appropriate adaptations and necessary extensions to our approach to deal with these classes of constraints. More precisely, in the first subsection we put forward the basis for handling dynamic constraints using just the object level. To overcome a number of difficulties encountered using just the object level, we propose in the second subsection a better solution by building a meta-layer for controlling integrity constraints. In the third subsection, we present different forms of constraints which may be handled by our approach.

3.1 Integrity Constraints as a CO-NETS Class

In what follows, first, we present the basis of our approach in handling integrity constraints. That is, we propose to associate with each CO-NETS component a corresponding ‘constraint’ class. Second, we relate ‘constraint’ transitions in this class with ‘method’ transitions in the corresponding CO-NETS on the basis of a new ‘synchronization’ inference rule. Third, we advocate the inherent consistency problems to be taken into account in order to make this solution meaningful.

Basic ideas for handling integrity constraints. As we just mentioned, the first ideas for dealing with integrity constraints in our approach is to associate with each component an ‘integrity constraint’ class (we use a class instead of a component as we consider no ‘integrity’ inheritance in this first step). The constituents of each constraint class are as follows.

- A ‘constraint’ place (instead of an ‘object’ place) : Tokens in this place are also records or tuples, precisely of the form $\langle \text{Constraint-identifier} \mid \text{inf}_1 : \text{val}_1, \dots, \text{inf}_k : \text{val}_k \rangle$. While **Constraint-identifier** is self-explanatory, the different pairs of ‘**information:value**’ represent the necessary information, either constant or changing, for expressing a given constraint (using associated transitions).
- With each constraint tuple, we associate at least one transition which reflects how different information in the tuple are related, and how they may change in a consistent way. In most cases we also use some ‘extra’ variables which have to be binded when we synchronize such constraint transitions with intrinsically dependable transitions in the corresponding CO-NETS component.

Example 5. Following these guidelines, in Figure 6 we have described two integrity constraints we want to associate with the **Account** component.

1. By the first tuple $\langle Cst1 \mid \text{Sum} : S, Hlimit : H, Llimit : L \rangle$ and its corresponding transitions represented by T_1 and T_2 , we want to specify the integrity constraint reflecting the fact that the sum of all account balances (in the account CO-NETS component) should neither exceed a certain (constant) value H nor go below L . This is what expresses the corresponding transitions T_1 and T_2 : we may always increase (resp. decrease) this total sum unless H (resp. L) is no exceeded (resp. reached).
2. The second constraint concerns the global interest (i.e the sum of account interest) which should not exceed a particular percent of the global sum of account balances. The transition T_3 reflects this fact. ■

Constraint enforcement using transitions synchronization. Given a constraint class, the question now is how to avoid the *violation* of such constraints. For this aim, the solution we propose is to relate each constraint transition with one or more corresponding methods (i.e. transitions)—from the associated component—which may violate it. We refer to such a binding as a synchronization relation and we denote it using ‘||’. The main idea under this synchronization

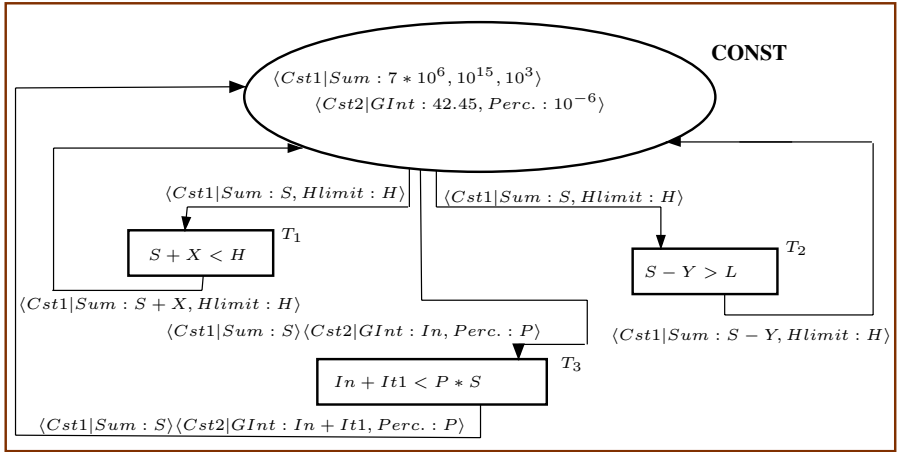


Fig. 6. Integrity Constraints for the CO-NETS Account Specification

is that a transition in a given component can be fired only and only if any intrinsically dependent constraint transition can also be fired. In other words, the method is allowed to take place if and only if any constraint affecting it is not violated. Before introducing the corresponding general inference rule that governs such a synchronization, let us explain this notion using the above example. □

Example 6. By assuming that in the constraint **Cst1** (in Figure 6) the attribute **Sum** contains effectively the sum of all account balances (the next paragraph addresses the realization of this assumption), then the constraint transition T_1 will not be violated if and only if it is fired simultaneously with any withdraw captured by the transition **WDR** in the Account CO-NETS component. In the same sense, the transition T_2 has to be fired in parallel with the method **DEP**. Finally, to ensure the constraint on interest, transition T_3 should be fired in parallel with the transition **INTR**. From this synchronization, three important consequences have to be deduced. Firstly, what we have called as extra variables in constructing constraint transitions should be substituted by appropriate variables from the corresponding methods. In this sense, the free variables X (resp. Y) in transition T_1 (resp. in T_2) corresponds to W (resp. D) in transition **WDR** (resp. **DEP**), and the variable $It1$ in transition T_3 corresponds to I in transition **INTR**. Of course we can directly use the same variables and avoid these substitutions. However, on the one hand we argue that it is more flexible to conceive constraints independently from CO-NETS component specification. On the other hand, as we sketched later, it is necessary to make such a separation if we want to allow also consistent transactions. Secondly, by making such a synchronization, integrity constraints are surely respected. In fact, either the two transitions fire and in this case all conditions are fulfilled, or none of them is fired due to a violation of some conditions in one of the two transitions. Thirdly, it is crucial to mention that each synchronization rule will now play the role of the semantics of the corresponding method. That is, in this example the described rewrite rules in section 2 for **WDR**,

DEP and INTR have to be *synchronized* with the corresponding rules associated with constraint transitions. ■

On the light of these explanations, the corresponding inference rule that we refer to as constraint synchronization rule can be formulated as follows:

Constraint synchronization rule: Given two rewrite rules

$$t_1 : |[b_1(x_1, \dots, x_n)]_{\otimes}| \Rightarrow |[b_2(x_1, \dots, x_n)]_{\otimes}| \in R_{comp} \text{ and}$$

$$t_2 : |[b_3(y_1, \dots, y_n)]_{\otimes}| \Rightarrow |[b_4(y_1, \dots, y_m)]_{\otimes}| \in R_{const}$$

Their synchronization, denoted by $t_1 \| t_2$ **where** $\bigwedge_i \{x_i/y_i\}$, is defined by:

$$\frac{[w_1] \Rightarrow [w'_1] \dots [w_n] \Rightarrow [w'_n] \dots [z_1] \Rightarrow [z'_1] \dots [z_m] \Rightarrow [z'_m]}{|[b_1(\bar{w}/\bar{x})]_{\otimes}| \Rightarrow |[b_2(\bar{w}'/\bar{x})]_{\otimes}| \wedge |[b_3(\bar{z}/\bar{y})]_{\otimes}| \Rightarrow |[b_4(\bar{z}'/\bar{y})]_{\otimes}| \wedge \bigwedge_i \{x_i/y_i\}}$$

Remark 2. As we mentioned, the part $\bigwedge_i \{x_i/y_i\}$ expresses the fact some variables from the constraint rules (i.e. y_i) have to be substituted by corresponding variables from methods (i.e. x_i). On the other hand, we have used two rewrite systems: R_{const} for representing rewriting rules governing transitions in the constraint class, whereas R_{comp} represents rules governing transitions of the corresponding component specification. This distinction is crucial because during the application of the above inference rule, (variable) instantiations of rules from R_{comp} are made with respect to the CO-NETS component current marking, whereas instantiations in R_{const} are made w.r.t. a constraint state which is a multiset of tuples representing the current constraint tuples in the place *CONST*. Finally, we note that $b(\bar{w}/\bar{x})$ stands for the replacement of some of the variables in (x_1, \dots, x_n) (abstracted here by \bar{x}) by appropriate terms from (w_1, \dots, w_n) (abstracted by \bar{w}). ♦

Example 7. Let us first recall the rewriting rules corresponding to R_{const} ; those of R_{comp} correspond exactly to the rules of the account component described in section 2.

T1: $(Const, \langle Cst1 \mid Sum : S, Hlimit : H \rangle)$
 $\Rightarrow (Const, \langle Cst1 \mid Sum : S + X, Hlimit : H \rangle)$ if $(S + X < H)$

T2: $(Const, \langle Cst1 \mid Sum : S, Llimit : L \rangle)$
 $\Rightarrow (Const, \langle Cst1 \mid Sum : S - Y, Llimit : L \rangle)$ if $(S - Y > L)$

T3: $(Const, \langle Cst1 \mid Sum : S \rangle \oplus \langle Cst2 \mid GInt : In, Perc. : P \rangle)$
 $\Rightarrow (Const, \langle Cst1 \mid Sum : S \rangle \oplus \langle Cst2 \mid GInt : In + It1, Perc. : P \rangle)$ if $(I + It < P * S)$

Following the intuitive explanation of necessary synchronizations presented in the above example, the rewrite rules that govern the account component in the presence of the above constraints are:

WRD_{T1}: $WDR \parallel T_1$ **where** W/X

DEP_{T2}: $DEP \parallel T_2$ **where** D/Y

Intr_{T3}: $INTR \parallel T_3$ **where** I/It ■

Constraints in presence of Transactions. In [AS00a], we have introduced how transactions as expressions on rewrite rule labels may be expressed using the CO-NETS approach. In some detail, transactions may be constructed using sequence of two or more rule labels (denoted by $;$), choice between some rules (denoted by $+$), parallel application of two rules (denoted by $|$), etc. As an example, we may have the transaction $WDR_1; DEP_1; WDR_2; (WDR_3 + DEP_2)$ that has to be performed without interference of other rules (ACID property). That is, we have to perform a withdraw followed by a deposit, followed by another withdraw and after then we have to choose between a withdraw or a deposit. In order to control the non violation of the above constraints (i.e. T_1 and T_2) we have to synchronize this transaction with these two constraints. The main problem consists, thus, in finding the adequate substitution to the ‘extra’ variables. Two possibilities should be distinguished depending the choice of WDR_3 or DEP_2 that we will distinguish them by the operator \vee . That is, in the presence of the two above integrity constraints, the following synchronization have to set in order to enforcement the above transaction

TRANS₁: $WDR_1; DEP_1; WDR_2; (WDR_3 + DEP_2) \parallel T_1 \parallel T_2$ **where** $(((W_1 + W_2 + W_3))/X \wedge D_1/Y) \vee ((W_1 + W_2)/X \wedge (D_1 + D_2)/Y)$

As given in this illustration to incorporate integrity constraints in any transaction, first, we should ‘index’ the attribute variables in different rules (for instance by the order in which a given rule appear in the transaction) in order to distinguish them. In this example, we have used W_1, W_2, D_1, \dots . Second, the choice between the two pairs of (X, Y) is systematically resolved by indefinite or free variables. Indeed, if for instance in the choice expression $(WDR_3 + DEP_2)$ the deposit rule is selected then the variable W_3 will be indefinite whereas the variable D_2 receives a specific value, and henceforth the second pairs of (X, Y) are selected to fire the transition T_1 and T_2 .

Related semantical problems. As it can be easily noticed, all the above constraints have no complete meaning without an adequate adaptation, on the one side, of the object creation / deletion process in a given component depicted in Figure 2. On the other hand, in order to allow introduction of integrity constraints at any time, and not obligatory at the creation of the system, we should also have an appropriate (net-) process for adding constraints which takes into consideration existing object instances.

Adaptation of the objects creation / deletion process. For objects creation / deletion in a given component, we have to adapt the net presented in Figure 2. This adaptation should allow updating object state dependent information in different constraints each time a new object is created or an existing object is deleted. That is, in this process in addition to the object place we should also consider the constraint place as input / output place with appropriate tokens inscribing its arcs.

Example 8. With respect to the two account constraints we are considering, Figure 7 presents how the of balance sum of all accounts as well as the global

interest (selected from the constraint place **CONST**) have to be accordingly increased (resp. decreased) each time we create (resp. delete) an object state. This update is captured by the two transition T_{cr} and T_{dl} respectively. We recall that the variables in_1, \dots, in_3 represent initial values (during the creation of an account).

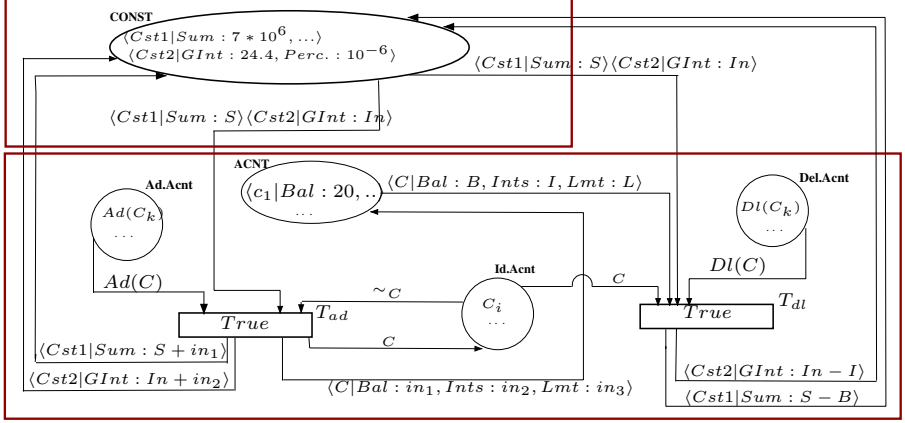


Fig. 7. Objects creation / deletion in presence of constraints

Finally, the two rewrite rules associated to this adapted objects creation / deletion process (in presence of integrity constraints) correspond to:

$$\begin{aligned}
 T_{cr}: & (Ad.Acnt, Ad(C)) \otimes (Const, \langle Cst1 | Sum : S \rangle \oplus \langle Cst2 | GInt : In \rangle) \otimes \\
 & (Id.Acnt, \sim C) \Rightarrow (Const, \langle Cst1 | Sum : S + In_1 \rangle) \otimes (Id.Acnt, C) \\
 & \otimes (ACNT, \langle C | Bal : in_1, Intr : in_2, Lmt : in_3 \rangle) \\
 & (Const, \langle Cst2 | GInt : In + in_2 \rangle) \\
 T_{dl}: & (Dl.Acnt, Dl(C)) \otimes (Const, \langle Cst1 | Sum : S \rangle \oplus \langle Cst2 | GInt : In \rangle) \otimes \\
 & (Id.Acnt, C) \otimes (ACNT, \langle C | Bal : B, Intr : I, Lmt : L \rangle) \\
 & \Rightarrow (Const, \langle Cst1 | Sum : S + B \rangle) \\
 & (Const, \langle Cst2 | GInt : In - I \rangle)
 \end{aligned}$$

Constraint creation in a running system. To allow introduction of constraints while the system is running, we should be able to update their constraint tuples, particularly those depending on existing objects. For this purpose we propose an appropriate net. The main ideas under such a process or net is to read (through a read-arc) *all* object attributes relevant to a given constraint and update its corresponding tuple. For traversing all relevant objects we propose to introduce an artificial (or temporary) place to control the already processed objects (by storing their identities). This construction is made explicit using our running example.

Example 9. For the two constraints we considered so-far, the net corresponding to their creation (at any time) is depicted in Figure 8. The elements of this

net may be highlighted as follows. The (initial) values of the attributes **Hlimit**, **Llimit** in the constraint **Cst1** as well as **Perc** in the constraint **Cst2** are fixed by the user (or properly by the Bank manager), whereas the changing attributes corresponding to **Sum** in **Cst1** and **GInt** in **Cst2** should be initially stated to zero. This initialization is expressed in the two tuples in the constraint place **Const**. Due to the fact that **Sum** and **GInt** have to be derived from existing objects at this moment, we propose for this purpose the transition **CRT-CST**. In this transition in addition to the input places **Const** and **ACNT** (with a read-arc), we have added a new temporary place, identified by **Temp.Id**, that is initially empty (i.e. with **nil** as depicted). This place allows traversing all objects without any duplication. In fact, when this transition is no more firable, it implies systematically that all objects have been traversed (because in this case the place **Temp.Id** will contain all object identities and by consequence the condition $\sim C$ will become no more true³). ■

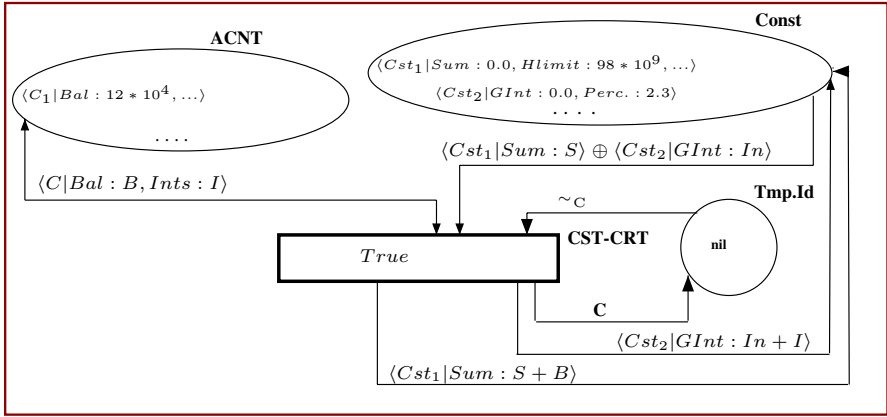


Fig. 8. Constraints creation in a consistent way

Finally, to close this subsection we survey this (first step of) proposal for handling integrity constraints associated with a given CO-NETS component.

Proposition 1. *The integrity constraints (restricted at present to cumulative ones) corresponding to a CO-NETS component specification are ensured under the following conditions:*

- These constraints are conceived as a class as presented above.
- Appropriate synchronizations should be established between rewrite rules corresponding to transitions in the constraint class and intrinsically dependent transitions in the CO-NETS component.
- Object creation / deletion in a given component has to be adapted in order to update constraint attributes.
- Created constraints in a running system have to be updated using an appropriate net.

³ Recalling that $\sim C$ means that C has not to be in the place **Temp.Id**.

3.2 Meta-level for a Flexible Consistency Management

Although the approach presented so far allows to effectively handle consistency in CO-NETS components, it presents several drawbacks. This makes it very difficult to be applied in a flexible way to complex components with several integrity constraints. Among the drawbacks we intend to overcome in this subsection, we list the following:

1. The adapted object creation / deletion process as well as the proposed net for creating constraints (while the system is running) are intrinsically related to the constraints we want to specify. In other words, they work only for specific constraints and not for any constraints. As a consequence of this rigidity, is that, with each introduction of a new integrity constraint we have to introduce two other new nets (one for object deletion / creation and one for constraint creation). These new nets obviously enter into conflict with the existing ones; because the creation / deletion of objects should be specified by just one net. A second way may consist in modifying the two nets each time new constraints are introduced. This however may lead to a very complex and hardly conceivable nets in the presence of several constraints.
2. The second less hard drawback concerns the constraint class itself which requires that each time we introduce a new constraint we have to manually introduce the corresponding transition with its different arc inscriptions. In fact, it would be more appropriate if we can avoid such transition constructions.

The purpose of this subsection is to overcome such drawbacks and result in an improved approach which is more flexible. Indeed, It is not difficult to deduce that the invoked shortcomings are mainly due to the absence of some meta-reasoning that *parameterizes* all elements to be changed each time a new constraint has to be modified, deleted or newly created. On the basis of this observation and from the fact that the proposed approach is based on three main constructions—namely the constraint class, the adapted object creation /deletion process, and the net for creating constraints—we propose in what follows for each construction a meta-level that makes it working for any existing or newly introduced constraints.

Meta-construction of constraint classes. As depicted in Figure 9, to allow introduction of integrity constraints without resorting to (manually) construct their corresponding transitions, the main idea is to replace all constraint transitions by just one but a *non-instantiated* transition. This transition, we denote by $Tr(i)$, receives its behaviour from the corresponding meta-place identified by **Meta-Const** through an appropriate read-arc. That is, following the same reasoning we proposed for the dynamic modification of component behaviour, each tuple in this meta place corresponds to a transition behaviour of a given integrity constraint. The semantics that allows propagating this behaviour to the object-level consists in a simplified version of the inference rule associated with the dynamic modification of behaviour. More precisely, we have the following inference rule:

Constraint Propagation Rule: For each non-instantiated rewrite, $Tr(i) : (Const, In) \Rightarrow (Const, Out)$ if $Cond$, we have:

$$\frac{\exists \sigma_i, \exists \sigma_o, \exists \sigma_c \wedge \langle Tr(n) \mid [\sigma_i(In)], [\sigma_o(Out)], [\sigma_c(Cond)] \rangle \in M(\mathbf{Meta-Const})}{Tr(n) : \sigma_i(In) \Rightarrow \sigma_o(Out) \text{ if } \sigma_c(Cond)}$$

Note that in the same spirit as for the inference rule for dynamic behaviour, **In**, **Out** are variables over multisets of constraint tuples, and **Cond** is a boolean variable over such multisets. On the other hand, we recall that $M(\mathbf{Meta-Const})$ represents the current marking of the place **Meta-Const**, and the natural **n** corresponds to an assignment to the variable **i** reflecting a precise transition.

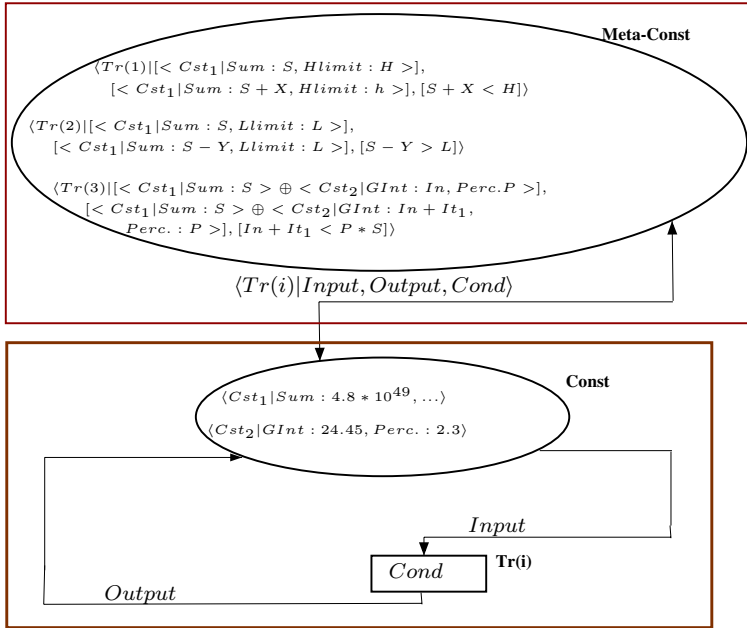


Fig. 9. Constraint class specification using a Meta level

□

In this way we can add any constraint without a need to draw its corresponding transition, while its behaviour is completely captured through the introduction of the corresponding token in the meta-place.

Example 10. In Figure 9 we have replaced the three transitions of the constraint class in Figure 6 by just one non-instantiated transition, whereas the behaviour of each of the three transitions is transformed into three corresponding tokens stored in the place **Meta-Const**. Of course, like for the dynamic evolution of components we have to add transitions for dynamically deleting, adding or updating a given constraint. ■

Meta-creation / deletion of objects with integrity constraints. In the same spirit, by using a meta-reasoning level the process of creation / deletion of objects will become compatible with any integrity constraint. More precisely, first, we keep unchanged all arc inscriptions which neither go to nor from the constraint place. In fact such inscriptions are independent from any constraint. Second, all other arc inscriptions are to be replaced by corresponding variables. Third, we built a new *meta-place* we identified by **Meta-CreatConst**, which has to contain the behaviour as tokens for these inscriptions. Moreover, in order to allow a transition to have more than one condition, we propose a more flexible token form in this meta-place. That is, each token in the meta-place is considered to be of the form:

□

$$\langle Tr_Id \mid Input, [Condition1, Output1], [Condition2, Output2], \dots \rangle (***)$$

In this way for each case (of the condition) the appropriate effect (as output token) is selected.

Example 11. The corresponding net for account object creation / deletion with the presence of constraints is depicted in Figure 10. In this net we have, for instance, included the two tokens—as behaviour explicitly described in Figure 7 through transitions T_{cr} and T_{dt} —of the two constraints we considered so-far.

□

Meta-constraint creation in a running system. Borrowing the same ideas of meta-reasoning, it is also possible to update (appropriate attributes of) any newly introduced integrity constraint without constructing a particular net, but just by adding an adequate behaviour as a token for such an update. In this process as depicted in Figure 11, tokens in the meta-place **Meta-creatconst** are represented using the same above structure (***), but in addition to the first component (i.e. Input) we should also introduce another element we denoted by **read**. This item represents the selected part of attributes from the objects place which is relevant to such update (for a particular constraint).

3.3 More Complex Integrity Constraints

After putting forward the basis ideas and their inherent theoretical underpinning by concentrating more on cumulative integrity constraints, the purpose of this subsection is to emphasize the generality of the approach in handling more complex constraints, including : (1) integrity constraints involving complex derived information ; (2) constraints including more than one component; and (3) constraints requiring the respect of particular life-cycles.

To emphasize the adequacy of the proposed approach in handling these subclasses of constraints, on the one hand, we use (a variant of) an example borrowed from [DB00]. In this example as depicted in (the low part of) Figure 12, we have two CO-NETS components : The employee and the department ones. On the other hand, to keep the presentation of the example more readable we deal just with the object level with its constraint classes. However, as we described, a more flexible version (but hardly readable) may systematically derived by constructing a meta-level for different constraints.

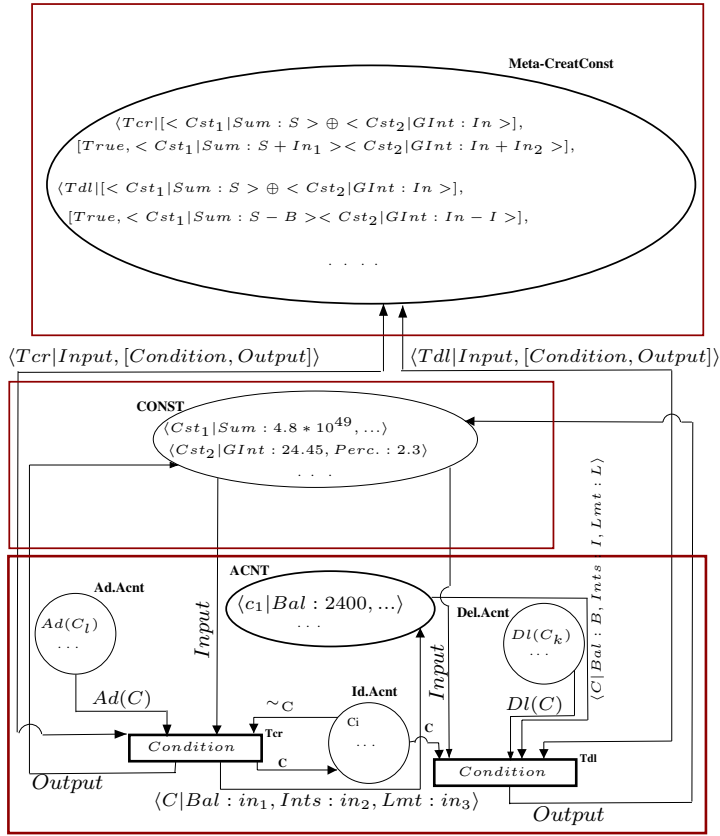


Fig. 10. A net with meta-places for creating /deleting objects with constraints

Description of the example. As we mentioned in this example we have two components: The employee component and the department component. As depicted in Figure 12, each employee is characterized by his/her salary (Sal), his/her department (Dep), and his/her position or specialization (Spec). The position has to be an element of the following list : [jun(ior), ana(lyst), prog(rammer), s(enio)r-an(alyst), m(ana)g(e)r]. The company may increase the salary of an employee (through the transition INC), it may punish an employee by decreasing its salary (through the transition PNS). Also through the transition PRM an employee may be promoted to a next specialization. For this aim we use the same specialization life-cycle as proposed in [DB00], that is, $jun \rightarrow ana \rightarrow sr_an \rightarrow mgr$ or $jun \rightarrow prog \rightarrow sr_pr \rightarrow mgr$. We also allow using the transition DPRM to de-promote a given employee (which represents the inverse of the promotion). In this component we also have a subclass modelling the managers, which may have in addition to the employee attributes, additional characteristics, like administrative responsibilities, etc. We note here that the managers are not concerned by the methods Promotion and

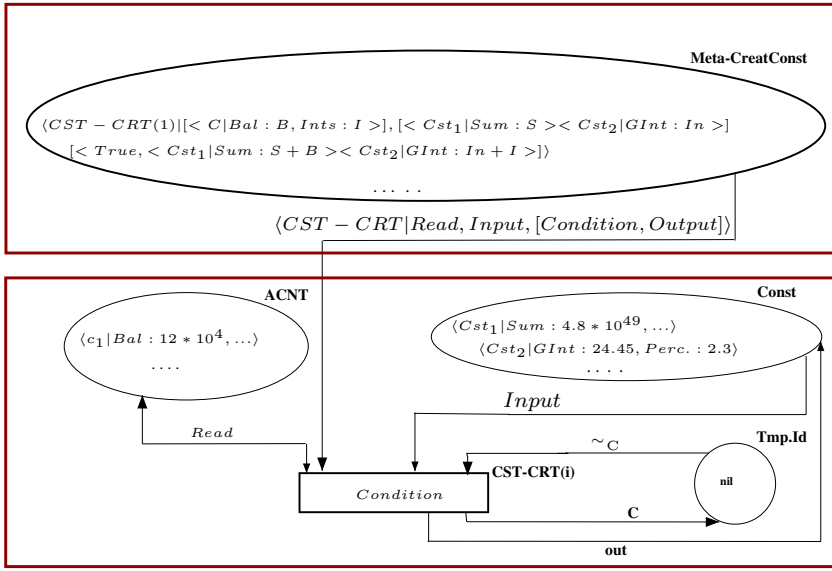


Fig. 11. A generic update process for runtime creation of constraints

De-promotion: This is expressed by adding to their two transitions input arcs going from the place **Id.Manager** with the inscription $\sim Id$. In the department component, each department is mainly characterized by its budget, etc.

Constraints on life cycles. As described the growth of the specialization of an employee must respect the explicitly given life cycle. The question is how to ensure the respect of such growth (or decrease in case of De-promotion). Following the proposed approach, we have to represent necessary information of this constraint as a tuple, associate it (a) corresponding transition(s), and relate such transition(s) with the promotion or De-promotion transition using an appropriate synchronization. Concerning the information to be in the tuple, we may easily notice that any life-cycle is an expression on elementary items (i.e. specializations) constructed using *sequence* and *choice* operators. This algebras can be straightforwardly specified using the following OBJ specification—of course other operators may be specified if necessary.

obj Life-cycle is

```

sort ITEM SEQ-ITEM CHOICE-ITEM LIFE-CYCLE.
subsort ITEM SEQ-ITEM CHOICE-ITEM < LIFE-CYCLE .
op Jun, ana, sr_an, sr_prmgr, prog : → ITEM .
op _;_ : LIFE-CYCLE LIFE-CYCLE → LIFE-CYCLE .
op _+_ : LIFE-CYCLE LIFE-CYCLE → LIFE-CYCLE .
vars R1, L, R, R0, L0, R1 : SEQ-ITEM
vars Q, Q' : ITEM
endo.
```



For our example the appropriate expression that reflect the proposed growing life cycle corresponds to:

$$Jun ; ((prog ; sr_pr) + ana ; st_sr)) ; mgr$$

For the inscription on the transition (represented in Figure 12 by LF), the ideas is to find a generic form which provides for a given (current specialization) item, denoted Q , the corresponding next item with respect to a given life cycle algebra. An analysis of the relation between an item and its corresponding next one (under the natural condition that each item appears only once in a life-cycle) results in the following generic form:

$$(R_i+)^*(R_j;)^*Q ; Q'(;R_k)^*(+R_l)^*\vee((R_i+)^*(R_j;)^*Q(+R_k)^* ; (R_k+)^*Q'(;R_k)^*(+R_l)^*$$

Globally the meaning of this generic form for extracting the next item from a current one may be highlighted as follows. If the current item (i.e. Q) is directly followed through a sequence by another one (i.e. we have a part in the life cycle including $Q ; Q'$) then the next item should be Q' . The remaining possibility is when the current item (i.e. Q) is the last in a given sequence then the next item should be the first one starting a new sequence (i.e. all choices following the sequence⁴ in which Q appears as last item have to be skipped).

Finally, after this conception it remains just to synchronize this constraint transition namely LF with the corresponding method transition PRM (or the de-promotion one $DPRM$), where we have used for simplicity the same variables namely Q and Q' .

Proposition 2. *Under the above construction for life cycle tuples and constraint transitions, the synchronization of such transitions with corresponding method transitions results exactly in the respect of this life cycle.*

The proof of this proposition may be derived directly from the above constructions.

Constraints involving views. The third subclass of integrity constraints we focus on in this paragraph concerns constraints which necessitate the computation of derived information and include comparisons of different information. A typical example using our company example is the fact that: “*the salary of the manager should be always greater that the salary of all other employees (in each department)*”. Before establishing the corresponding constraint tuple as well as the associated transition(s) for ensuring such constraint, first, we need the greater or maximal salary among all employees, we denote it by **SalMax**. Indeed, by assuming that this constraint is respected at the moment of its creation (as we developed in the above subsections) this constraint is intuitively respected by stating that in case of increasing this greater salary it should not exceed the corresponding one of the manager, and in case of decreasing the manager salary it should not go below this greater salary. This is exactly what is

⁴ We use the notation $(R_i+)^*$ or $(+R_i)^*$ (here the choice operator is at the beginning) to express the choice of severals (or none) sequences and $(R_i+)^+$ for a choice including at least one sequence.

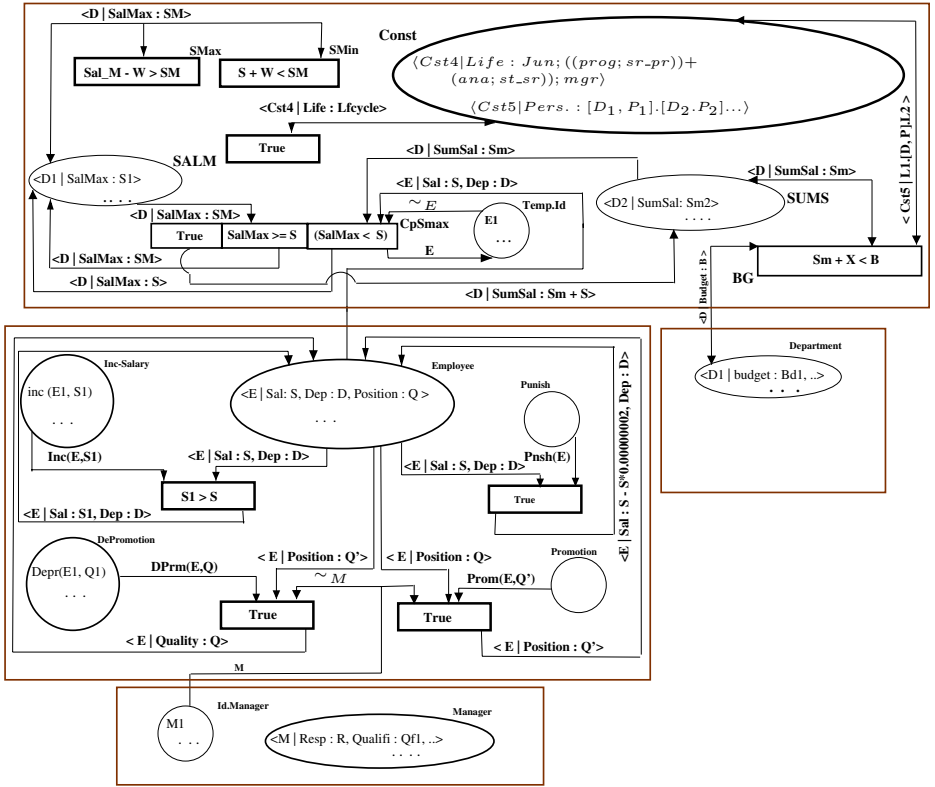
expressed by the constraint transitions **SMax** and **SMin**. Moreover, in this case there is no need for a constraint tuple. In fact, all needed information are either extracted from the temporary place **SALM**, which recuperates at each time the employees' greater salary, or from the methods to be synchronized with these constraints (i.e. decreasing the manager salary for **SMax** and increasing the employee salary for the constraint transition **SMin**). The computation of this greater salary using the transition **CpSmax** presents no difficulties, that is, we have just to traverse all employee states and compare the current salary with the temporary maximal salary. In order to avoid duplication we have to use a temporary place for recording the processed employee states. Besides the computation of this greater salary, we have also used this transition to generate the sum of all employee salaries in each department (we gather in the place **SUMS**).

Constraints involving more than one component. Finally, we sketch here an example of constraints including more than one component. The constraint we consider that relates the component employee with the department one is the following : *“The sum of all employee salaries of a given department should not exceeds certain percent of the budget affected to such a department”*. For handling this constraint, first, we need the computation of the sum of all salaries in a given department as a derived information. This sum as we just mentioned is already computed using the transition **CpSmax** and the place **SUMS**. Second, as a constraint tuple we need to store the percent associated with each department. This can be done using a list of pairs (department, percent) of the form $[dep1, perc1].[dep2, perc2]...$, as indicated in the constraint tuple **Cst5**. Third, using the constraint transition **BG** we establish such a constraint. Finally a synchronization has to be made between this constraint transition and the method transition for increasing the salary.

4 Conclusions

In this paper we proposed an extension of the CO-NETS approach for expressing and enforcing integrity constraints in runtime evolving concurrent information systems. The proposed approach also allows such integrity constraints to be manipulated (i.e. created, modified or deleted) while the specified system is still running. The key ideas for managing consistency in the CO-NETS approach consist, first, in associating with each CO-NETS component specification a corresponding constraints class. Each constraint tuple contains necessary information as (special) ‘object’ attributes with transitions reflecting the allowed change of such informations. Second, we propose to synchronize such transitions with component methods which may intrinsically alter them. Finally, to allow a runtime management of such constraints, we have adapted the concepts and constructions dealing with dynamic modification in CO-NETS. Moreover, we showed how our approach to consistency includes several known subclasses of constraints.

However, to take into account other relevant classes of constraints, a lot of work remains ahead. Among others, as further extensions of this proposal we plan to particularly deal with dynamic constraints involving history of actions nicely



Abbreviations:

$$\text{Lfcycle} : (R_i +)^* (R_j)^* Q ; Q' (; R_k)^* (\neg R_l)^* \vee (R_i +)^* (R_j)^* Q (+ R_k)^* ; Q' (; R_k)^* (+ R_l)^*$$

Fig. 12. A simple company specification as a CO-NET with different forms of constraints

handled using the (past-) temporal setting [Saa91, GL96, CKS96]. Adapting the work in [MT99] could be a best starting point towards such an extension. Indeed in this proposal, the authors relate rewrite logic—the semantical framework of CO-NETS—with event structures as a semantic framework to object specification [ECSD98].

Acknowledgments. We are grateful to the anonymous referees for helpful comments that improved this presentation. We also acknowledge the comments received from the editors of this volume.

References

- [Aou00] N. Aoumeur. Specifying Distributed and Dynamically Evolving Information Systems Using an Extended CO-Nets Approach. In G. Saake, K. Schwarz, and C. Türker, editors, *Transactions and Database Dynamics*, volume 1773 of *Lecture Notes in Computer Science, Berlin*, pages 91–111. Springer-Verlag, 2000. *Selected papers from the 8th International Workshop on Foundations of Models and Languages for Data and Objects, Sep. 1999, Germany*.
- [AS99] N. Aoumeur and G. Saake. Towards an Object Petri Nets Model for Specifying and Validating Distributed Information Systems. In M. Jarke and A. Oberweis, editors, *Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering, CAiSE'99*, volume 1626 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, 1999.
- [AS00a] N. Aoumeur and G. Saake. An Appropriate Semantics for Distributed Active Object-Oriented Databases on the Basis of the CO-NETS Approach. In *International Conference on Software Engineering Applied to Networking and Parallel/ Distributed Computing (SNPD '00)*, Reims, France , 2000.
- [AS00b] N. Aoumeur and G. Saake. Cooperative Information Systems Modelling and Validation Using the CO-NETS Approach: The Chessmen Making Shop Case Study. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proc. of the 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*, Berlin, Germany, pages 361–383, 2000.
- [AS00c] N. Aoumeur and G. Saake. Specifying and Validating Train Control Systems Using an Appropriate Component-Based Petri Nets Model. In *Proc. of the Petri Nets in Design, Modelling and Simulation of Control Systems Special session at IFAC Conference CSD2000, Bratislava, Slovakia*, 2000.
- [CKS96] S. Conrad, H. Klein, and K. D. Schewe. Integrity in Databases: 6th International Workshop on Foundations of Models and Languages for Data and Objects. Preprint Nr. 4, Fakultät für Informatik, Universität Magdeburg, 1996.
- [DB00] E.O. De Brock. A General Treatment of Dynamic Integrity Constraints. *Data & Knowledge*, 32:223–246, 2000.
- [ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 6, pages 167–198. Kluwer Academic Publishers, Boston, 1998.
- [GL96] M. Gertz and U.W. Lipeck. Deriving Optimized Integrity Monitoring Triggers from Dynamic Integrity Constraints. *Data & Knowledge*, 20:163–193, 1996.
- [GWM⁺92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, 1992.
- [MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems : Specification*. Springer-Verlag, New York, 1992.
- [MT99] J. Meseguer and C. Talcott. A Partial Order Event Model for Concurrent Objects. In *Proc. of CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 415–430. Springer-Verlag, 1999.

- [PS98] M. P. Papazoglou and G. Schlageter, editors. *Cooperative Information Systems: Trends and Directions*. Academic Press, Boston, 1998.
- [RSSS91] K. A. Ross, D. Srivastara, P. J. Stuckey, and S. Sudarshan. Foundation of Aggregation Constraints. *Theoretical Computer Science*, 193:47–74, 1991.
- [Saa91] G. Saake. Descriptive Specification of Database Object Behaviour. *Data & Knowledge*, 6:47–73, 1991.
- [SCT01] G. Saake, S. Conrad, and C. Türker. Evolving Objects: Conceptual Description of Adaptive Information systems. In H. Balsters, B. De Brock, and S. Conrad, editors, *Database schema Evolution and Meta-Modeling: 9th International Workshop on Foundations of Models and Languages for Data and Objects (FOMLADO/DEMM 2000)*, Dagstuhl, Germany, Lecture Notes in Computer Science Vol. 2065, pages 162–180, Springer-Verlag, 2001.
- [SL90] A.P. Sheth and J.A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

Adaptive Specifications of Technical Information Systems

Sören Balko

Otto-von-Guericke-Universität Magdeburg
Institut für Technische und Betriebliche Informationssysteme
Postfach 4120, D-39016 Magdeburg, Germany
balko@iti.cs.uni-magdeburg.de

Abstract. Nowadays formal specification techniques have become popular in the development process of many kinds of software systems. Since many technical scenarios are based on computer control, their software implementations also depend on previously established formal descriptions. Additionally, technical information systems such as production control facilities often have a long life span. Due to this fact, dynamic changes become increasingly likely. For example, these changes may be induced by introduction of new laws, altered production goals, human interactions, or any other kind of external influences. However, these changes sometimes require an alteration of the software. To fit the implementation, its formal specification (if present) has to be adapted appropriately. Ordinary specification methods do not permit a post-implementation change in the specification itself but rather an afresh specification effort throwing away the current formal description. Since the necessary changes would frequently result in minor adaptations in the specification, this situation is very unsatisfactory. To avoid or reduce this re-specification effort, we are working on extensions of established specification techniques which can cover adaptive specifications.

The remainder of this paper is organized as follows. Section 1 introduces our project and presents a simple classification of adaptive specifications. In Section 2 we briefly present the main issues of the case study and motivate adaptive specifications. Subsequently we suggest some syntactical extensions of TROLL in Section 3. And finally, Section 4 gives an outlook on future work in this project.

1 Motivation

The project *Semantics of Adaptive Workflows*¹ (SAW) deals with extensions to workflow specifications. The extensions shall introduce some specification dynamics. That is, following an environmental change, the specification of an information system must provide adaptation techniques. The main idea is to replace small portions of the formal specification by updated parts. By means of these techniques the overall specification and reengineering efforts shall be reduced heavily.

¹ Supported by the DFG (Deutsche Forschungsgemeinschaft): grant no. Sa 465/19-1.



Our first approach is based on the object-oriented specification language TROLL. In [5] corresponding syntactical extensions were proposed (*dy*TROLL). Some proposals to describe the formal semantics of adaptive specifications can be found in [4]. TROLL is supplemented by the graphical **s**pecification language OMTROLL [6]. For the sake of simplicity we classify workflow adaptations by means of OMTROLL state diagrams:

- **Simple Adaptation:** In this case, the number, identity, and sequence of possible *states* remains equal. However, *guards* (preconditions) and *events* may be subject of changes. In Figure 1 three simple adaptation types are distinguished:
 1. *Condition Adaptation:* The guard of a state transition is altered.
 2. *Event Adaptation:* The event of a state transition is changed.
 3. *Condition-Event Adaptation:* Represents the combination of the previous two.

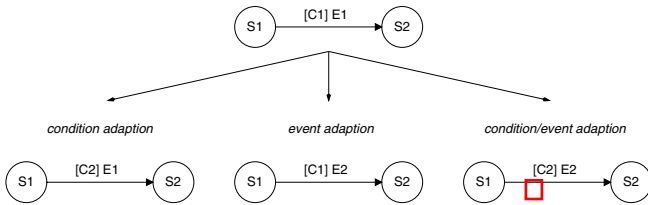


Fig. 1. Simple Adaptation Types

- **Complex Adaptation:** Complex adaptation allows states to be added, replaced, and removed. In Figure 2 complex adaptation types are depicted:

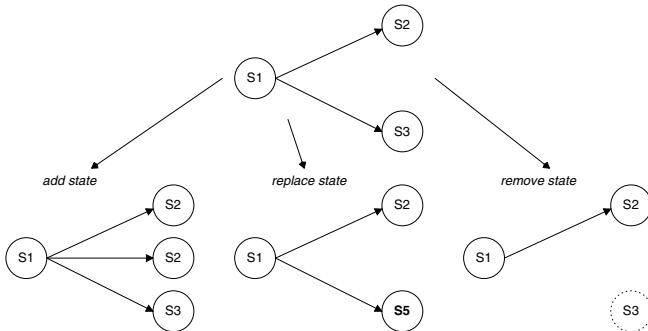


Fig. 2. Complex Adaptation Types

2 Case Study

In our project we formally specified a case study which deals with a production environment. Within this scenario certain work pieces shall be treated by three different machines (M_1 , M_2 , M_3) in a fixed order. Each **MACHINE** consists of two local buffers which contain the unprocessed and processed work pieces. In addition to the machines there is also an entry storage (**IN**) acting as the source for the unprocessed work pieces. Furthermore, a global exit storage (**OUT**) assimilates the completely processed work pieces.

The main characteristic of the scenario is that there is no physical connection among the machines and storages. The transport of the work pieces shall be carried out by so called *Holonic Transport Systems* (**HTS**) instead. The **HTS** are mobile robots which communicate by radio broadcast. One HTS can carry at most one work piece at a time. Additionally, there exists no central control for the scenario but every HTS acts independently. Every time a machine requests a transport of work pieces, the HTS must negotiate the order *with each other*.

Obviously, there is a wide range of opportunities to specify this scenario. In this paper we focus on one adaptation scenario: the *job initiation strategy*. The job initiation strategy applies to the machines and determines the way in which the machines request the transport of a work piece. Clearly, there are at least two job initiation strategies:

- **always:** The machines initiate new jobs whenever it is possible, i. e., whenever the local entry buffer has space for a work piece or the local exit buffer is not completely empty.
- **urgent:** In this opposite case, the machines initiate new jobs only if they are blocked, i. e., whenever the local entry buffer is empty or the local exit buffer is full.

Obviously, the presented *job initiation strategies* are contradictory. This means that no machine may obey both strategies equally. Since we can easily imagine further alternatives, it would also not make sense to put all these strategies into the original specification. In most cases it would be even impossible to predict *all* possible specification adaptations. To ease a quick adaptation to a new strategy during the system's lifetime, the formal specification must therefore allow a flexible exchange of dynamic specification parts. This way, the approach presented in [3] is put into practice. □

3 Adaptive Specifications

In this section we propose some syntactical extensions of the TROLL specification language. We will therefore reuse some former approaches [7] to split a TROLL specification into a *rigid* (i. e. fixed) and a *dynamic* (i. e. changeable) part. The main idea is to have the main characteristics of the system specified in the rigid part. Everything that is about to be altered shall be exchangeable and, therefore, resides in the dynamic part. Since a complete specification of the scenario is rather extensive, here we will only present a part of the machine

specification which illustrates the adaptation. In Figure 3 the OMTROLL state diagram of the machines is depicted.

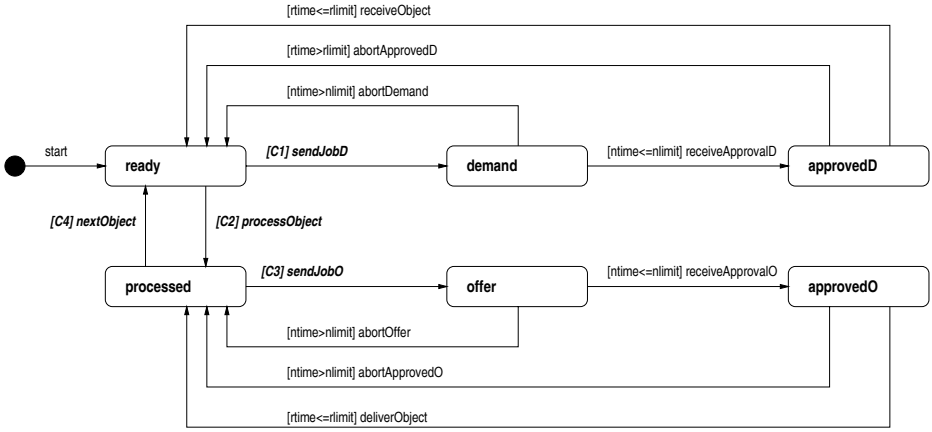


Fig. 3. State Diagram of Class **MACHINE**

By this specification, each machine processes a work piece (**processObject**), takes the next one (**nextObject**) and so on. To get new work pieces delivered, the event **sendJobD** requests all HTS to negotiate this job. Following, the machine waits until a prefixed time has elapsed ($\text{ntime} > \text{nlimit}$) and sends a new request or an approval from one HTS (**receiveApprovalD**) is received. Subsequently, the machine again waits ($\text{rtime} > \text{rlimit}$) for the work piece delivery (**receiveObject**). The proceeding in the case of the machine offering a work piece is similar. To distinguish between the two job initiation strategies different preconditions ($C_1 \dots C_4$) have to be applied to **sendJobD**, **sendJobO**, **processObject**, and **nextObject**. In Table 1 these preconditions are depicted.

Table 1. Condition Adaptation: Job Initiation Strategies

guard	<i>always</i>	<i>urgent</i>
C_1	$\text{InBuf.available} < \text{InBuf.capacity}$	$\text{InBuffer.available} = 0$
C_2	$\text{InBuf.available} = \text{InBuf.capacity}$	$\text{InBuf.available} > 0$
C_3	$\text{OutBuf.free} < \text{OutBuf.capacity}$	$\text{OutBuf.free} = 0$
C_4	$\text{OutBuf.free} = \text{OutBuf.capacity}$	$\text{OutBuf.free} > 0$

On the basis of the state diagram (and other class diagrams not depicted here) a TROLL specification can be generated. To allow an adaptation of the job initiation strategies the corresponding condition adaptation has to be put into the dynamic specification.

```

object class MACHINE
  identification ByMid:(mid)
  attributes
    mid:nat constant.
    rtime:nat initialized 0.
    ntime:nat initialized 0.
    rlimit:nat initialized 100.
    nlimit:nat initialized 20.
    job:|JOB| initialized nil.
    hts:|HTS| initialized nil.
  components
    InBuf:SOURCE
    OutBuf:DESTINATION
    WorkPiece:WORKPIECE
  events
    start birth
    calling
      {InBuffer.available==0} sendJobD,
      {InBuffer.available>0} processObject.

    sendJobD
      changing
        ntime:=0,
        job:=job.create(DEMAND,self)
      calling
        foreach h:HTS
          do h.receiveJob(job) od,
          abortDemand.

    abortDemand
      enabled
        ntime>nlimit
      calling
        sendJobD,
        processObject.

    receiveApprovalD(h:|HTS|,j:|JOB|,offer:real)
      enabled
        sometime sendJobD
        sincelast receiveApprovalD
        and j=job
      changing
        hts:=h,
        rtime:=0,
        rlimit:=offer
      calling
        abortApprovedD.

    abortApprovedD
      enabled
        rtime>rlimit
      calling
        sendJobD,
        processObject.

    receiveObject(o:|OBJECT|)
      enabled
        sometime receiveApprovalD
        sincelast sendJobD
      calling
        sendJobD,
        processObject.

    processObject
      calling
        nextObject,
        sendJobD.

    sendJobD
      OutBuf.free:=0
      changing
        ntime:=0,
        job:=job.create(OFFER,self)
      calling
        foreach h:HTS
          do h.receiveJob(job) od,
          abortOffer.

    abortOffer
      enabled
        ntime>nlimit
      calling
        sendJobD,
        nextObject.

    receiveApprovalO(h:|HTS|,j:|JOB|,offer:real)
      enabled
        sometime sendJobD
        sincelast receiveApprovalO
        and j=job
      changing
        hts:=h,
        rtime:=0,
        rlimit:=offer
      calling
        abortApprovedO.

    abortApprovedO
      enabled
        rtime>rlimit
      calling
        sendJobD,
        nextObject.

    deliverObject(!o:|OBJECT|)
      enabled
        sometime receiveApprovalO
        sincelast sendJobD
      calling
        sendJobD,
        nextObject.

    nextObject
      calling
        processObject,
        sendJobD.

  axioms
  JobInitiation
    initialized
    {
      sendJobD
        enabled
          InBuf.available = 0.
        processObject
          enabled
            InBuf.available > 0.
      sendJobD
        enabled
          OutBuf.free = 0.
      nextObject
        enabled
          OutBuf.free > 0.
    }

  mutators
  replaceStrategy(Add:set(axiom),
    Rem:set(axiom))
  enabled Rem  $\subseteq$  JobInitiation
  changing
    JobInitiation:=JobInitiation-Rem
    JobInitiation:=JobInitiation $\cup$ Add.

end object class

```

To obtain a dynamic specification we may name one or even several **axioms**. Unlike usual attributes, here, specification formulas build the values. In this example, we only have one axioms attribute (**JobInitiation**). To modify the axioms attribute, a particular sort of events which are called **mutators** is needed. For example, we may add and remove axioms or exclude the complete dynamic specification. Since we exclusively exchange the strategies, a mutator **replaceStrategy** is sufficient for this purpose.

The main idea in this approach is to have a simple (but complete!) specification described by the static part. Any kind of adaptations can be specified in the dynamic part. Therefore, special *axioms* contain the currently valid specification. To modify these attributes, we may specify arbitrary *mutators*. Analogously to usual events these mutators may be constrained by guarding conditions, they may change the value of axiom attributes and they may also call other mutators.

4 Conclusions

In this short paper we presented the goals of the SAW project. A main goal is, syntactically and semantically extending established specification methods to cover adaptive specifications. For the time being, this extension has been conducted in TROLL by means of examples. The overall idea is to separate the specification into rigid and dynamic parts. The dynamic part is not fixed but may rather be modified by so-called mutators during lifetime.

In the future we will particularly deal with an extension of our approach to other specification languages. Furthermore, we will try to base the formal semantics on a common basis. Additionally, the complete case study will be examined with respect to adaptation scenarios. In [2] this brief presentation of the adaptive case study was presented more thoroughly. Furthermore, in [1] a new approach to transfer OMTROLL diagrams into petri nets was illustrated by means of this case study.

References

1. N. Aoumeur, S. Balko, and G. Saake. Towards a Three-Level Methodology for Developing Cooperative Information Systems. Preprint Nr. 5, Fakultät für Informatik, Universität Magdeburg, 2000.
2. S. Balko. Adaption von Auftragsvergabestrategien in der Referenzfallstudie Produktionstechnik. Preprint Nr. 1, Fakultät für Informatik, Universität Magdeburg, 2001.
3. G. Saake, C. Türker, S. Conrad. Evolving Objects: Conceptual Description of Adaptive Information Systems. In H. Balsters, B. De Brock, and S. Conrad, editors, *Database schema Evolution and Meta-Modeling — 9th International Workshop on Foundations of Models and Languages for Data and Objects (FOMLADO/DEMM 2000)*, Dagstuhl, Germany, Lecture Notes in Computer Science Vol. 2065, pages 162–180, Springer-Verlag, Berlin, 2001.
4. S. Conrad, J. Ramos, G. Saake, and C. Sernadas. Evolving Logical Specification in Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 7, pages 199–228. Kluwer Academic Publishers, Boston, 1998.

5. S. Conrad, G. Saake, and C. Türker. Towards an Agent-Oriented Framework for Specification of Information Systems. In J.-J.C. Meyer and P.-Y. Schobbens, editors, *Formal Models of Agents (ESPRIT Project ModelAge Final Report)*, Lecture Notes in Computer Science Vol. 1760, pages 57–73, Springer-Verlag, Berlin, 2000.
6. G. Denker and P. Hartel. *Troll – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics*. Informatik-Berichte, Technische Universität Braunschweig, 1997.
7. C. Türker, S. Conrad, and G. Saake. Dynamically Changing Behavior: An Agent-Oriented View to Modeling Information Systems. In J. Fiadeiro and P.-Y. Schobbens, editors, *Proc. of the 2nd Workshop of the ModelAge Project (ModelAge'96), Sesimbra, Portugal, January 1996*, pages 257–270. Departamento de Informatica, Universidade de Lisboa, 1996.

Evolving the Software of a Schema Evolution System^{*}

Kajal T. Claypool, Elke A. Rundensteiner, and George T. Heineman

Worcester Polytechnic Institute, 100 Institute Road,
Worcester, MA, USA
{kajal, rundenst, heineman}@cs.wpi.edu
<http://davis.wpi.edu/dsrg>

Abstract. The object model represents the core of an OODB system. Any change in the object model such as the addition of an association or aggregation relationship affects many sub-systems including the schema evolution system. Under the current tightly-coupled database architecture, updating the object model is an extremely expensive undertaking for a database vendor both in terms of time and resources. Adding a new construct to the object model impacts the schema evolution system in two ways: (1) the new construct requires a new set of schema evolution primitives to enable its evolution; and (2) existing schema evolution primitives must be modified to assure that they conform to the new constraints of the new object model. One traditional approach to address this is to manually change all affected software, a time consuming task. We present an alternate two-prong solution. We first decouple the constraints from the schema evolution primitives and secondly we provide a mechanism that allows for the declarative definition of both the primitives and the constraints. We show via examples that we can reduce the software evolution cost of the schema evolution component completely for semantic extensions to the object model and can partially reduce the cost for most other new modeling constructs.

Keywords: Software Evolution, Loosely coupled Architecture, Schema Evolution

1 Introduction

In recent years much energy has been vested in modeling languages and their expressibility in terms of designing and modeling complex applications. This increase in the expressive power of modeling languages such as the Unified Modeling Language (UML)[Boo94] has resulted in an impedance mismatch with current database technology used to persistently store information for such UML

^{*} This work was supported in part by the NSF NYI grant #IRI 94-57609. We would also like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and Informix for software contribution. Kajal T. Claypool would like to thank GE for the GE Corporate Fellowship.

modeled applications. An *Aggregation* relationship [Boo94], a semantic variation of an association relationship, is just one example of a UML construct not supported by any OODB system [Obj93,Tec94,Tec92]. Many applications today thus need to seek alternative hard-coded, customized implementations of the non-database constructs to support their UML model.

A more attractive option is to have support at the database level to avoid redundant and duplication of effort as well as to assure correct behavior. However, extending the underlying object model of an existing OODB to support these UML constructs is challenging. Any change in the object model forces an update of all database sub-systems that depend on the object model. These systems include the Schema Repository and the Schema Evolution facility to name a few, representing typically a significant software evolution. Under the current tightly-coupled database architecture, the norm for practically all commercial systems [Obj93,Tec92,Tec94], this is an extremely expensive undertaking for a database vendor both in terms of time and resources. And hence most database vendors would be challenged to provide such support in a timely fashion.

In this paper, we focus on reducing the cost of evolving the software for the schema evolution manager of an OODB system, a subsystem heavily impacted by any change in the object model. A change in the object model is manifested in the schema evolution system in two forms: (1) a new set of schema evolution primitives is required to enable evolution of the new construct and (2) modifications to the existing schema evolution primitives to assure that they conform to the new constraints of the new object model. Our approach looks at providing a cheaper and easier to maintain mechanism(s) for accomplishing the above tasks as an alternative to the otherwise required manual software evolution. To the best of our knowledge while many researchers have looked at schema evolution of a static object model, no one has looked at the software evolution of a schema evolution facility itself or alternatives to it in the event of a change to the object model.

The key to our approach is (1) an alternative mechanism to the hard-coded schema evolution primitives for specifying schema evolution primitives, (2) the de-coupling of the constraints from the schema evolution primitives, and (3) an extensible framework to support new evolution operations over time. We have developed SERF [CJR98], an extensible schema evolution framework that allows users to specify SERF transformation templates, arbitrarily complex schema evolution operations. SERF transformation templates combine existing system-defined schema evolution operations with OQL to provide this flexibility. Thus, SERF templates can be used as a flexible mechanism for formulating new schema evolution operations for a semantically extended object model. Here no addition to the schema evolution subsystem is necessary to make it conform to the changed object model.

Furthermore, to enable de-coupling of constraints from the schema evolution primitives, we introduce the notion of *contracts* [Mey92] for SERF templates. A SERF template with contracts formulates a *SERF wrapper* for the existing schema evolution primitives where the contracts specify the constraints that were originally hard-coded into the schema evolution primitive. From a software engineering perspective, the idea of replacing programming with declarative con-

structs is not new. However, a DBMS is an uncharted domain with respect to this as: (1) there is a canonical architecture constructed from standard subsystems; (2) there is a clear behavior and responsibilities for each subsystem; (3) the merger of object-oriented modeling and persistent storage will make it difficult for any OODB vendor to keep up. Alternatively, any OO model application will find it very inefficient to constantly flatten their models to be stored in persistent databases. Our system provides this extensibility at a relatively cheap price. Developing the pre- and post-conditions declaratively is much easier than reprogramming a system. These contracts then can be verified prior to the execution of the schema evolution primitive thereby assuring that the newly added template will not compromise the consistency of the database. Using a SERF wrapper for schema evolution primitives now allows us to accomplish upgrade of evolution primitives in the event of a changed set of invariants simply by declaratively adding or modifying the contracts.

To support our hypothesis, in this paper we present two examples. In the first example, *association* relationship, we can accomplish only the update to existing schema evolution primitives using the SERF wrapper. In the second example, *aggregation* relationship, we show that both the update to existing primitives as well as composition of new primitives to support the evolution of aggregation relationships can be accomplished via SERF wrapper technology. Thus, in this case we completely eliminate the traditional software evolution cost for the schema evolution manager.

Outline. In Section 2 we present SERF and show how it can be utilized to provide new schema evolution primitives for new object model constructs. Section 4 introduces the contracts that we now propose as an addition to the SERF template and shows how we can accomplish the de-coupling of constraints from the code. Section 5 gives a complete example for a SERF Wrapper. Section 6 we present a summary of the benefits of using SERF for evolving the schema evolution software. Section 7 outlines relevant related work while we conclude in Section 8.

2 SERF: Providing Basic Schema Evolution Operations

The conventional approach for upgrading the schema evolution facility is the specification of new hard-coded schema evolution primitives. To eliminate the time costs involved in providing these at the system level we now present SERF [CJR98], an alternative mechanism for specifying the traditional hard-coded system-defined schema evolution.

Unlike current OODB systems that restrict schema evolution to a *predefined* set of basic schema evolution operations with *fixed* semantics [BKKK87, Tec94, BMO⁺89, Inc93, Obj93], the SERF framework addresses the limitation by allowing new schema evolution operations to be defined. The SERF framework enables *new*, *customizable* and possibly *very complex* schema evolution operations such as *merge*, *inline* and *split* [Ler96, Bré96] without forcing developers to directly modify database internal code. Moreover, for each transformation type itself there can be different semantics based on user preferences and application

needs. For example, two classes can be merged through a union, an intersection, or a difference of their attributes. Similarly, deleting a class that has a super-class and several sub-classes can be accomplished by either: (1) propagating the delete of the inherited attributes through all sub-classes; (2) moving the attributes up to the super-class; (3) moving the attributes down to all the sub-classes; or (4) selective composition of the three strategies.

Our approach is based on the hypothesis that complex schema evolution transformations can be broken down into a sequence of basic evolution primitives, where each basic primitive is an invariant-preserving atomic operation with fixed semantics provided by the underlying OODB system. To effectively combine these primitives and to perform arbitrary transformations on objects within a complex transformation, we use the standard query language OQL [Cat97]. We have already demonstrated that a language such as OQL is sufficient for accomplishing schema evolution [CJR98].

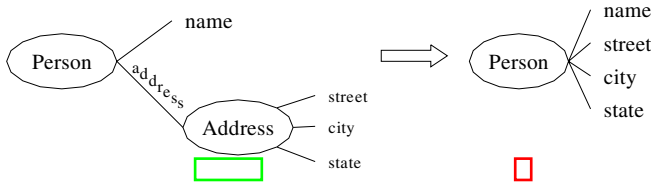


Fig. 1. Example of an Inline Transformation.

Consider for example, *inline*, replacing a referenced type with its type definition [Ler96], as shown in Figure 1. Here the **Address** type is inlined into the **Person** class, i.e., all attributes defined for the **Address** type (the referenced type) are now added to the **Person** type. Figure 2 shows the *inline* transformation expressed as a SERF transformation using OQL. The complex operation *inline* is thus broken down into a sequence of operations that can be expressed using the system-defined schema evolution primitives, such as **add-attribute**, OQL and system-defined update methods. For example, **Step A** in Figure 2 adds the attributes **Street**, **City** and **State** to the class **Person** by using the system-defined schema evolution operation **add-attribute**. **Step B** performs an OQL query to gather the extent of the class **Person**. In **Step C** we iterate over this extent and perform object transformations to copy the values of the attributes **person.address.Street**, **person.address.City** and **person.address.State** to the respective attributes in the class **Person**. The last step **Step D** finally deletes the attribute **address** from the class **Person**.

SERF Template. The SERF transformation as given in Figure 2 allows *flexibility*, it is not generalizable like the pre-defined evolution primitives of an OODB system. Thus, such a transformation can not be applied to any application schema other than the one for which it was defined. For example, the *inline* transformation shown in Figure 2 is valid only for the given classes **Person** and **Address**.

```

// Add the required attributes to the Person class
add_attribute (Person, Street, String, " ");
add_attribute (Person, City, String, " ");
add_attribute (Person, State, String, " ");

// Get all the objects for the Person class
define extents() as
    select c
    from Person c;

// Update all the objects
for all obj in extents():
    obj.set (obj.Street, valueOf(obj.address.Street));
    obj.set (obj.City, valueOf(obj.address.City));
    obj.set (obj.State, valueOf(obj.address.State))

// Delete the address attribute
delete_attribute (Person, address);

```

} Step A
 } Step B
 } Step C
 } Step D

Fig. 2. Inline Transformation Expressed in OQL with Embedded Evolution Primitives.

SERF transformations are thus not usable in their current form as a mechanism to add new schema evolution operations under object model changes. To remove this limitation, SERF uses templates, a SERF transformation that has been encapsulated and generalized via the use of the ODMG Schema Repository, a name and a set of parameters. As an example, Figure 3 shows a template corresponding to the transformation in Figure 2. The section of the template marked **Step E** shows the steps required to achieve the effect of **Step A** in Figure 2 in a general form. **Step C** is generalized in a similar fashion. **Step B** and **Step D** remain unchanged. Thus when this inline template shown in Figure 3 is instantiated with the parameters **Person** and **address** it produces the SERF transformation in Figure 2. Hence, newly added schema evolution operations can be executed by a user similar to the system-provided schema evolution primitive.

Application: Schema Evolution Primitives for Aggregation Relationship. Now consider adding the *aggregation* construct [Boo94] into an object model that already supports the *association* construct. Aggregation is a semantic extension of an association and provides an ownership constraint on an association; the UML representation of aggregation is shown in Figure 4. The evolution primitive for this can be formulated by providing ownership constraints for the low-level schema evolution primitives already defined for an association. These semantically-extended aggregation primitives can be expressed using a SERF template.

Figure 5 shows an example of the *aggregation* SERF template that creates an *aggregation* relationship between two classes. Here we use the evolution primitive *add-reference-attribute* [CRH99] to first create a uni-directional relationship between the two classes. We assume that when a new construct is added, the database is updated with new storage data structures and the

```

begin template inline (className, refAttrName)
{
    refClass = element (
        select a.attrType
        from MetaAttribute a
        where a.attrName = $refAttrName
        and a.classDefinedIn = $className; )

    define localAttrs(cName) as
        select c.localAttrList
        from MetaClass c
        where c.metaClassName = cName;

    // get all attributes in refAttrName and add to className
    for all attrs in localAttrs(refClass)
        add_atomic_attribute ($className, attrs.attrName,
                               attrs.attrType, attrs.attrValue);

    // get all the extent
    define extents(cName) as
        select c
        from cName c;

    // set: className.Attr = className.refAttrName.Attr
    for all obj in extents($className):
        for all Attr in localAttrs(refClass)
            obj.set (obj.Attr, valueOf(obj.refAttrName.Attr));

    delete_attribute ($className, $refAttrName);
}

end template

```

} Step E

Fig. 3. The Inline Template.

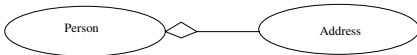


Fig. 4. A Sample Schema Showing Aggregate Relationship.

```

form-aggregation-relation ( Cs, r,
Cd, default)
{
    add-reference-attribute-
primitive
    (Cs, r, Cd, default);
    upgrade-to-aggregation ( Cd,
r );
}

```

Fig. 5. A Template For Creating an Aggregate Relationship Between Two Classes

system dictionary is updated. Thus, we use the system dictionary function **upgrade-to-aggregation** to enforce aggregation semantics on an association relationship: we inform the OODB system to maintain the aggregation semantics for this relationship. Similarly we can write templates for deleting and modifying the *aggregation* relationship.

Utilizing Templates. Thus, SERF templates provide a mechanism for combining existing system-defined schema evolution primitives to provide schema evolution operations for new constructs in the object model. These schema evolution operations for the new constructs can be collected into a **template library** and can be utilized by the users of the OODB system in identical fashion to the currently provided system-defined schema evolution primitives. An implementation of SERF, OQL-SERF, has been developed at Worcester Polytechnic Institute and has been demonstrated at SIGMOD [RCL⁺99].

3 Updating Existing Evolution Primitives - The Problem

Conventionally, schema evolution primitives contain hard-coded constraints that parallel the invariants of the object model. These constraints must be satisfied in order to guarantee the consistency of the system. Changes in the object model, such as adding an *association* relationship construct, changes these invariants. To guarantee the consistency of a schema evolution system, all existing schema evolution primitives must be updated to reflect the changed invariants. This is an expensive process that may require extensive re-engineering of the affected software.

For example, current OODB systems often do not differentiate between reference attributes and literal attributes and hence the evolution primitives do not differentiate between them either. However, as we show in this section it is not sufficient to simply treat them as two separate entities and provide extra evolution support for them; we need to closely re-examine the existing evolution primitives to determine how they are impacted. In this section we examine the consistency problems that can occur today in current OODB systems and in Section 4 we present a solution to the problem using the SERF framework. To highlight some of the consistency problems that can occur we augment the object model with association relationships.

In all of the scenarios that we have examined, when an object model is extended, the core functionality of existing evolution primitives itself is unaffected. However, the constraints that need to be checked to determine when these primitives can be applied can be greatly changed. Consider for example the *delete-class*(C_i) evolution primitive [PS87]. This primitive can only be applied when the class C_i is a *leaf* class (refer to Figure 6), i.e:

$$sub(C_i) = \emptyset \quad (1)$$

However, while this is a necessary and sufficient constraint for deleting of the **HomeAddress** class specified in the schema depicted in Figure 7, it is no longer a sufficient stipulation for a schema that contains relationships as in Figure 8.

For example, deleting the *leaf* class **Address** in the schema in Figure 8, a valid evolution operation, causes dangling references and hence compromises the consistency of the system by violating both the structural integrity (schema-level) and the referential integrity (object-level) of the system. The *delete-class* primitive must be re-implemented with the constraint that a class cannot be

```

boolean delete-class (Class c)
{
  if (c.subclasses().count() == 0 )
  {
    delete all objects of c
    destroy the class c
    return true;
  }
  else return false;
}

```

Fig. 6. Pseudo-Code for *delete-class* Primitive

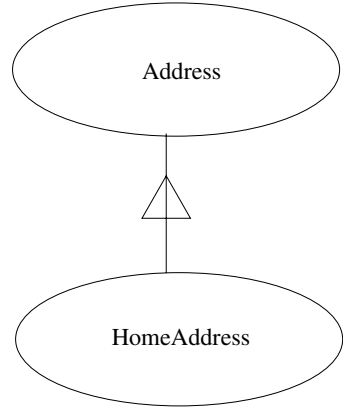


Fig. 7. An Example Schema Showing No Relationships

deleted if it has other classes referring to it. Using the notation in Table 2 this could be expressed as:

$$in - degree(C_i) = 0 \quad (2)$$

However, while the conditions in Equations 1 and 2 ensure the structural integrity of the schema, they still cannot ensure the referential integrity. Consider for example the schema shown in Figure 9. In this example, the class **Person** has a direct relationship with the class **Address**, while the class **Home-Address** is inherited from the class **Address**. The class **Person** and all its subclasses **Student** and **Teaching-Assistant** inherit the relationship to the class **Address**. However, when instantiating the class **Person** or any of its subclasses it is possible at the object level to instantiate a relationship with an object of the type **Home-Address** rather than an object of type **Address**. Thus, while the conditions in Equations 1 and 2 ensure that *delete-class* (**Home-Address**) does not violate the structural integrity of the system, we could potentially violate the referential integrity of the system.

To capture consistency violations at the object level, we thus define a third constraint for the *delete-class* primitive that must hold before the deletion of a class can occur:

$$\forall o_i \in extent(t) : obj - in - degree(o_i) = 0 \quad for \quad t = type(C_i) \quad (3)$$

The constraints expressed in Equations 1, 2, and 3 together now ensure the consistency of the database both in terms of the structural as well as the referential integrity when the primitive *delete-class* is executed.

Today, while most state-of the art OODB systems allow the use of reference attributes, the *delete-class* primitive in these systems only needs to satisfy one

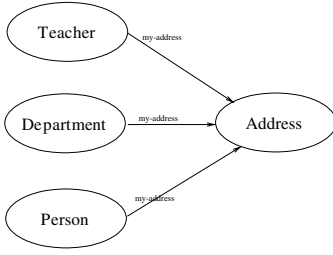


Fig. 8. An Example Schema Showing Relationships

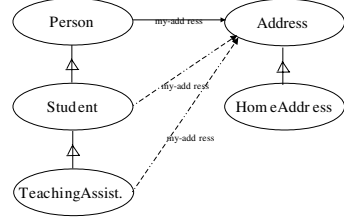


Fig. 9. A Sample Schema Containing Relationships via Inheritance

constraint, i.e., the class being considered for deletion must be a leaf class. From the example shown here it can be seen that this, even in current systems, can cause inconsistencies. The *delete-class* primitive is just one example and the same holds true for all of the other schema evolution primitives.

In light of this analysis, we now present the set of invariants that guarantee the consistency of the object model in the presence of relationships. Table 1 presents the invariants for the ODMG object model with relationships.

Table 1. Invariants of the Model

Axioms	Description
Rootedness	$T = \mathbf{root} \mid \forall t \in \mathbf{types}(\mathcal{C}), t \in \mathit{sub}^*(T)$
Closure	$\forall t \in \mathbf{types}(\mathcal{C}), \mathit{super}^*(t) \in \mathbf{types}(\mathcal{C}) \mid t = \mathbf{root}$
Pointedness	$\perp = \mathbf{leaf} \mid \mathit{sub}(\perp) = \emptyset$
Nativeness	$N(t) =$ The set of native (local) properties of type t
Inheritance	$H(t) =$ The set of inherited properties of type t
Distinction	$c \in \mathcal{C} \mid c$ is unique
Degree	total in-degree (T-IN) / total out-degree (T-OUT) is an invariant

4 Contracts: De-coupling the Constraints

One of the major drawbacks of current schema evolution software is the tight coupling of the constraints, (the invariants) with the actual behavior of the evolution primitive. This tight coupling results in heavy costs when an update to the existing software becomes necessary. Thus, the second step to our approach is de-coupling the constraints from the actual implementation code of the schema evolution operations. To accomplish this we introduce the notion of *contracts*, a declarative mechanism for expressing the constraints within the SERF frame-

work. A SERF template with contracts is termed as a *SERF Template Wrapper*¹. Changes to the invariants of the object model now merely result in the update of the declarative contracts associated with the evolution operations rather than the update of the actual system code. In this section we introduce *contracts* and show how the de-coupling of constraints can be achieved.

Contracts provide a declarative description of the behavior of a template (or primitive) as well as a mechanism for expressing the constraints that must be satisfied prior to the execution of the actual evolution primitive. Contracts are divided into two categories: **pre-conditions** and **post-conditions**.

The constraints, termed *pre-conditions*, are placed prior to any body of template code (OQL statement including system-defined schema evolution primitive). The *pre-conditions* are separated from the actual OQL statements by means of the keyword **requires** and are expressed using the functions and notation shown in Table 2. On the other hand, *post-conditions*, a set of contracts that appear after the body of the actual schema evolution operation at the end of the SERF template, specify the behavior of the primitives. These post-conditions are preceded by the keyword **ensures** and describe the exact changes that are made to the schema by the evolution operator, and hence its behavior. Figure 10 shows the post-conditions for the **delete-class** primitive.

Beyond de-coupling the constraints from the schema evolution primitives, the *pre-conditions* and *post-conditions* offer an additional advantage of behavior verification. Using a pre-execution verification process such as theorem proving we can disable the execution of schema evolution primitives that may either fail during execution or cause inconsistencies in the database. This can be a significant advantage for improving the execution performance of a schema evolution operation.

Example: Updating Existing Primitives to Reflect Addition of Association Relationship. Consider adding associations to the object model of an OODB system. An upgrade to the schema evolution facility requires new schema evolution primitives to handle the creation, modification, and deletion of uni-directional and/or bi-directional associations. This upgrade cannot be circumvented and hence new schema evolution primitives must be added to the system. However, all existing schema evolution operations must be updated to conform to the new set of invariants. For example, to delete a class prior to the existence of associations, the constraint that a class needed to be a *leaf* class was necessary to ensure that the resulting schema and database was consistent, i.e., the **delete-class** preserved the database consistency. With the addition of associations, this constraint alone is not sufficient. We now also need to ensure that the **to-be-deleted** C_i class is not referred to by another class. Moreover, no objects in the database must refer to the objects of the C_i class. So while the conditions that need to be enforced prior to the execution the schema evolution operation have to be upgraded, the actual actions of the operations do not change. Hence the evolution primitive **delete-class** itself does not change.

¹ We use the terms template and wrapper interchangeably from here on.

Table 2. Notation for Axiomatization of Schema Changes

Term	Description
$\text{types}(\mathcal{C})$	The set of all types in the system
s, t, T, \perp	Elements of $\text{types}(\mathcal{C})$
$\text{super}(t)$	The set of all direct supertypes of type t
$\text{sub}(t)$	The set of all direct subtypes of type t
$\text{super}^*(t)$	The set of all direct and indirect supertypes of type t
$\text{sub}^*(t)$	The set of all direct and indirect subtypes of type t
$\text{in-paths}(t)$	The set of all paths $\langle c, r \rangle$ referring to type t , i.e., all the types in the system that are referring to the type t
$\text{in-degree}(t)$	The count of all paths referring to type t , i.e., the number of elements in $\text{in-paths}(t)$
$\text{out-paths}(t)$	The set of all paths $\langle t, r \rangle$ going out of type t , i.e., all the references that are made to other types by the type t
$\text{out-degree}(t)$	The count of all paths going out of type t , i.e., the number of elements in the set $\text{out-paths}(t)$
$\text{obj-in-degree}(o_i)$	The number of objects referring to the object o_i
$\text{obj-out-degree}(o_i)$	The number of objects being referred to by the object o_i
\mathcal{R}	The set of all relations in the system
$N(t)$	The native (local) properties of type t
$H(t)$	The inherited properties of type t

Figure 10 shows the modified constraints after the addition of an association relationship for the *delete-class* primitive as pre-conditions². Thus, in this model it is easy to extend or modify the constraints without re-writing the code for the evolution primitive. While we could not completely eliminate the effort of evolving the schema evolution subsystem, we have demonstrated that we can reduce the update cost.



5 An Example - Creating New Primitives with Contracts

We now describe a complete example for writing new schema evolution primitives using SERF template wrappers for adding *aggregation* relationships. Here we show how the ability of SERF to compose new primitives as illustrated in Section 2 and the contracts shown in Section 4 can be combined to provide a solution. The *delete-class* primitive and its pre-conditions as given in Figure 10 are no longer sufficient for handling the deletion of a class C_i that has an *aggregation* relationship with another class C_j . The *delete-class* primitive now needs to propagate the delete of an *aggregator* to all of the *aggregated* classes.

² The notation used here is a set-theoretic version of the contract language.

```

delete-class (  $C_i$  )
{
  requires:
     $C_i \in \mathcal{C} \wedge$ 
     $\sigma(C_i) \in \mathbf{types}(\mathcal{C}) \wedge$ 
     $sub(C_i) = 0 \wedge$ 
     $in-degree(C_i) = 0 \wedge$ 
     $\forall o_i \in \mathbf{extent}(t)$ 
       $obj-in-degree(o_i) = 0$ 

  template body here

  ensures:
     $C_i \notin \mathcal{C} \wedge$ 
     $\sigma(C_i) \notin \mathbf{types}(\mathcal{C}) \wedge$ 
     $\forall \langle C_x, r_x \rangle \in out-paths(C_i)$ 
       $(\langle C_i \rangle \notin in-paths(C_x)) \wedge$ 
     $\forall C_x \in super(C_i)$ 
       $(C_i \notin sub(C_x))$  □
}

```

Fig. 10. Pre- and Post-Conditions for **delete-Class** Primitive in Contractual Form

In Figure 11, we show a delete template that correctly propagates a delete request to the *aggregated* classes. In this template wrapper, we first downgrade the aggregation relationship to a referential relationship using the system provided function **downgrade-aggregation**. The evolution primitive **delete-reference-attribute** deletes all the downgraded aggregation relationships and the **delete-class** template then deletes all aggregated classes themselves³. The final step to delete the aggregator itself is accomplished by the last invocation of the **delete-class** template. In all of these cases we make use of the **delete-class** template rather than directly invoking the **delete-class-primitive** to utilize the contracts defined for the **delete-class** wrapper.

6 Classification of Software Evolution Support

In this section we summarize the software evolution support offered by our system in an effort to reduce the re-engineering costs. A schema evolution system must provide for any supported modeling construct: (1) primitives to evolve the basic construct; (2) consistency of the system under evolution; and (3) as a desired feature it may provide more complex evolution of the supported constructs.

³ There is a possibility of failure of the **delete-class** template for an aggregated class as it is possible that the aggregated class participates in a relationship with some other class. However for simplicity we ignore this situation.

```

delete-aggregator (  $C_i$  )
{
    requires:
         $C_i \in \mathcal{C} \wedge$ 
         $\sigma(C_i) \in \mathbf{types}(\mathcal{C}) \wedge$ 
         $sub(C_i) = 0 \wedge$ 
         $in-degree(C_i) = 0 \wedge$ 
         $\forall o_i \in \mathbf{extent}(t)$ 
             $obj-in-degree(o_i) = 0 \wedge$ 
             $self-agg-degree(C_i) = 0$ 

     $agg-List = \text{select } c.agg\text{-paths from } c \text{ in MetaClass where } c.name$ 
     $= C_i$ 

    for all  $X$  in  $agg-List$ 
        downgrade-aggregation( $X$ );
        delete-reference-attribute ( $C_i, X.refAttr$ );
        delete-class( $X.className$ );

    delete-class( $C_i$ );

    ensures:
         $C_i \notin \mathcal{C} \wedge$ 
         $\sigma(C_i) \notin \mathbf{types}(\mathcal{C})$ 
         $\forall \langle C_x, r_x \rangle \in out\text{-paths}(C_i)$ 
             $\langle C_i \rangle \notin in\text{-paths}(C_x) \wedge$ 
         $\forall \langle C_x, r_x \rangle \in agg\text{-paths}(C_i)$ 
             $\langle C_i \rangle \notin in\text{-paths}(C_x) \wedge$ 
         $\forall C_x \in super(C_i)$ 
             $C_i \notin sub(C_x) \wedge$ 
}

```

Fig. 11. Template for Handling the Deletion of an Aggregator.

When new modeling constructs are added to the system, the schema evolution facility must be evolved to satisfy at least the first two of the above listed features. These new additions to the object model fall in one of the two categories (1) a completely new modeling construct, such as an *association* relationship; and (2) an enhancement (constraint addition) on an existing modeling construct, such as an *aggregation* relationship when an *association* relationship is already supported by the existing facility.

In the absence of a facility such as ours (SERF), the re-engineering costs for any category of change (as given above) is:

$$C = 3 + 3n \cdot p + 3n \cdot c \quad (4)$$

where n is the total number of existing modeling constructs, p is the number of evolution primitives and c is the number of complex evolution primitives

supported for each modeling construct. We assume that the re-engineering cost for update is 3 units, 1 unit for physically updating the primitive and 2 units for testing the system.

SERF Cost for New Construct. Using the SERF facility the cost of evolving the schema evolution facility when a completely new construct is added is given as:

$$C = 3 + 1n \cdot p + 1n \cdot c \tag{5}$$

Here the cost of updating the existing set of basic and complex primitives is reduced by 2 units to indicate that only the declarative constraints need to be updated. The primitives or the schema evolution facility however do not need to be re-compiled. The initial cost of 3 units indicates that the basic primitives for the modeling construct need to be added to the schema evolution facility and tested.

Thus, while the initial cost of adding new primitives remains the same, the SERF system provides a 3-fold savings for evolving a schema evolution facility when a completely new modeling construct is added.

SERF Cost for a Constraint Addition. Using the SERF facility the cost of evolving the schema evolution facility when new constructs are a constraint update of existing constructs is given as:

$$C = 1 + 1n \cdot p + 1n \cdot c \tag{6}$$

Here the cost of updating the existing set of basic and complex primitives is again reduced by 2 units to indicate that only the declarative constraints need to be updated. Moreover, the basic primitives for the construct can also be coded declaratively using SERF thereby reducing the cost of adding new primitives to 1 unit. Thus, the SERF system provides a 3-fold reduction for evolving the software of a schema evolution system when the new construct is a constraint enhancement of an existing construct.

7 Related Work

Schema Evolution. The goal of schema evolution research is to develop mechanisms to change not only the schema but also the underlying objects to have them conform to the modified schema. Most research and commercial systems today provide schema evolution in the form of a fixed set of evolution primitives [BKKK87,Inc93,Obj93,Tec94,BMO⁺89,SZ86,LH90]. Breche [Bré96] and Lerner [Ler96] have investigated more complex schema evolution operations such as *inline* and *merge*; in our previous work we show how the SERF system can be used to express schema evolution operations in a flexible and customizable fashion[CJR98]. In this work, we have taken the next step and shown how SERF

can be utilized to provide a loosely-coupled architecture for the schema evolution facility in order to ease the software evolution process of the same in the event of object model changes.

This set of schema evolution operations parallels the modeling constructs supported by the object model. However, to the best of our knowledge no one has researched the impact of a changing object model on an existing schema evolution facility. In this work we look at the issue of extensibility of schema evolution facilities in the light of a changing object model.

Extensible Systems. Hürsch et al. [HS96] have proposed a framework that captures the dependency between different subsystems in a schema and code. When a change occurs this dependency framework is used to formulate propagation patterns to maintain behavioral consistency. They use propagation patterns as a mechanism for maintaining programs. We propose here a similar approach. However, we utilize a declarative approach for specifying the constraints embedded within schema evolution primitive code. We utilize the notion of Contracts [Mey92] as first proposed by B. Meyer to specify the constraints in a declarative fashion. Formal verification [GSW95,ORS92,Bla98,GM93] techniques or more informal verification algorithms can be utilized to verify the contracts.

8 Conclusion

To summarize, in this paper we have presented a mechanism for reducing the time and cost of software evolution of a schema evolution facility in the light of object model changes. Our solution provides a mechanism (1) for specifying schema evolution operations and (2) for updating the constraints of the system-defined, hard-coded evolution primitives, without requiring additional coding or compilation. This system would not require the same update time and expense in the event of a change in the object model compared to the traditional cost involved in updating the tightly-coupled schema evolution facilities today. We also hypothesize that when the object model is augmented by a new construct, our framework can only be used to update the existing schema evolution primitives. New primitives for the evolution of the construct have to be added to the system in this case. However, in case the new construct is a semantic-extension of an existing construct, SERF can be utilized to provide both the new schema evolution primitives for its evolution as well as for updating the existing primitives.

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research.

References

- [BKKK87] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, pages 311–322, 1987.

- [Bla98] P.E. Black. *Axiomatic Semantic Verification of a Secure Web Server*. PhD thesis, Brigham Young University, February 1998.
- [BMO⁺89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.
- [Boo94] G. Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings Publications, 1994.
- [Bré96] P. Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.
- [Cat97] Cattell, R.G.G and et al. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.
- [CJR98] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Int. Conf. on Information and Knowledge Management*, pages 314–321, November 1998.
- [CRH99] K.T. Claypool, E.A. Rundensteiner, and G.T. Heineman. Extending Schema Evolution to Handle Object Models with Relationships. Technical Report WPI-CS-TR-99-15, Worcester Polytechnic Institute, March 1999.
- [GM93] M.J.C Gordon and T.F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, 1993.
- [GSW95] Y. Gurevich, N. Soparkar, and C. Wallace. Formalizing database recovery. In *COMAD*, 1995.
- [HS96] W.L. Hürsch and L.M. Seiter. Automating the Evolution of Object-Oriented Systems. In *International Symposium on Object Technologies for Advanced Software*, pages 2–21, Kanazawa, Japan, March 1996. Springer Verlag, Lecture Notes in Computer Science.
- [Inc93] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.
- [Ler96] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, Computer Science Department, 1996.
- [LH90] B. S. Lerner and A. N. Habermann. Beyond schema evolution to database reorganization. In *OOPSLA*, pages 67–76, 1990.
- [Mey92] B. Meyer. Applying “Design By Contract”. *IEEE Computer*, 25(10):20–32, 1992.
- [Obj93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems*. Object Design Inc., December 1993.
- [ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *11th CADE, Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.
- [PS87] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *OOPSLA*, pages 111–117, 1987.
- [RCL⁺99] E.A. Rundensteiner, K.T. Claypool, M. Li, L. Chen, X. Zhang, C. Nataraajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Restructuring Facility. In *Demo Session Proceedings of SIGMOD’99*, pages 568–570, 1999.

- [SZ86] A. H. Skarra and S. B. Zdonik. The Management of Changing Types in an Object-Oriented Databases. In *Proc. 1st OOPSLA*, pages 483–494, 1986.
- [Tec92] Versant Object Technology. *Versant User Manual*. Versant Object Technology, 1992.
- [Tec94] O₂ Technology. *O₂ Reference Manual, Version 4.5, Release November 1994*. O₂ Technology, Versailles, France, November 1994.

Schema Evolution and Versioning: A Logical and Computational Characterisation

Enrico Franconi¹, Fabio Grandi², and Federica Mandreoli²

¹ University of Manchester, Dept. of Computer Science, Manchester, UK
`franconi@cs.man.ac.uk`

² Università di Bologna, CSITE-CNR and DEIS, Bologna, Italy
`{fgrandi,fmandreoli}@deis.unibo.it`

Abstract. In this paper we study the logical and computational properties of schema evolution and versioning support in object-oriented databases. To this end, we present the formalisation of a general model for an object base with evolving schemata and define the semantics of the provided schema change operations. We will then sketch how the encoding of such a framework in a suitable Description Logic will allow the introduction and solution of interesting reasoning tasks at global database and single schema version levels.



1 Introduction

Schema evolution and versioning problems have been considered in the context of long-lived database applications, where stored data were considered worth surviving changes in the database schema [26]. According to the definitions given in a consensual glossary [21], a database supports *schema evolution* if it allows modifications of the schema without the loss of extant data; furthermore, it supports *schema versioning* if it allows the querying of all data by means of any schema version, according to the user or application preferences. With schema versioning, different schemata can be identified and selected by means of a suitable “coordinate system”: symbolic labels are often used in design systems to this purpose, whereas proper time values are the elective choice for temporal applications [14,15]. For the sake of brevity, schema evolution can be considered as a special case of schema versioning where only the current schema version is maintained.



In this paper, we present a formal approach, which has been introduced and analysed in [13], for the specification and management of schema versioning in the general framework of an object-oriented database, and discuss its logical and computational characteristics. The adoption of an object-oriented data model is the most common choice in the literature concerning schema evolution, though schema versioning in relational databases [11] has also been studied deeply. The approach is based on:

- the definition of an extended object-oriented model supporting evolving schemata, provided with all the usually considered schema changes, whose semantics is formalised;

- the formulation of interesting reasoning tasks (e.g. concerning database consistency), in order to support the design and the management of an evolving schema;
- an encoding, which has been proved correct, in a suitable Description Logic, which can then be used to solve the tasks defined for the schema versioning management.

Within such a framework, the main problems connected with schema versioning support will be formally characterised, both from a logical and computational viewpoint, leading to the enhancements listed in the following.

- The complexity of schema changes becomes potentially unlimited: in addition to the classical schema change primitives (a well-known comprehensive taxonomy can be found in [4]), our approach enables the definition of complex and articulated schema changes.
- Techniques for consistency checking and classification can be automatically applied to any resulting schema. We consider different notions of consistency:
 - *Global Consistency*, related to the existence of a legal database (or single class) instance for the evolving schema;
 - *Local Consistency*, related to the existence of a legal database (or class) instance for a single schema version.
- Classification tasks we define include the discovery of implicit inclusion / inheritance relationships between classes ([5]). Decidability and complexity results are available for the above mentioned tasks in our framework [13] and tools based on Description Logics can be used in practice.
- The process of schema transformation can be formally checked. The provided semantics of the various schema change operations makes it possible to reduce the correctness proof of complex sequences of schema changes to solvable reasoning tasks.

However, our semantic approach has not thoroughly addressed the so-called *change propagation* problem yet, which concerns the effects of schema changes on the underlying data instances. In general, change propagation can be accomplished by populating the new schema version with the results of queries involving extant data connected to previous schema versions. Moreover, from a theoretical point of view, dealing with the presence of object identifiers (OIDs, which correspond to real and conceptual objects in the “real world”) represents a non-trivial problem for the definition of such a query language, which, thus, must be very carefully designed. In Section 4, our proposal will be reviewed in the light of previous approaches concerning object languages dealing with OIDs (e.g. [1,19,20,10]), and directions for future developments will also be sketched.

The paper is organised as follows. Section 2 introduces the syntax and the semantics of an object-oriented model for evolving schemata support. Section 3 formally defines and exemplifies reasoning problems which are relevant for the design and the management of an evolving schema and analyses their computational complexity. In particular, Section 3.1 mentions a provably correct encoding of the object-oriented model for evolving schemata into a Description Logic,

showing that theoretical and practical results from the Description Logic field can be applied in our framework. After a survey of the current status of the field, a critical discussion (Sec. 4) about the proposed approach will precede the conclusions (Sec. 5).

2 The Data Model

In this Section we summarise a general object-oriented model for evolving schemata which supports the taxonomy usually adopted for schema changes, as first proposed in [13]. To this end, we will first formally introduce the syntax and semantics for a schema (version) and for the supported schema changes, and then formulate some interesting reasoning problems and analyse their computational properties.

2.1 Syntax and Semantics

The object-oriented model we propose allows for the representation of multiple schema versions. It is based on an expressive version of the “snapshot” – i.e., single-schema – object-oriented model introduced by [1] and further extended and elaborated in its relationships with Description Logics by [8,9]; in this paper we borrow the notation from [8]. The language embodies the features of the static parts of UML/OMT and ODMG and, therefore, it does not take into account those aspects related to the definition of methods. At the end of section 3.1 suggestions will be given on how to extend even more the expressiveness of the data model, both at the level of the schema language for classes and types and at the level of the schema change language.

The definition of an evolving schema \mathcal{S} is based on a set of class and attribute names ($\mathcal{C}_{\mathcal{S}}$ and $\mathcal{A}_{\mathcal{S}}$ respectively) and includes a partially ordered set of schema versions. The initial schema version of \mathcal{S} contains a set of class definitions having one of the following forms:

$$\begin{array}{l} \text{Class } C \text{ is-a } C_1, \dots, C_h \text{ disjoint } C_{h+1}, \dots, C_k \text{ type-is } T \\ \text{View-class } C \text{ is-a } C_1, \dots, C_h \text{ disjoint } C_{h+1}, \dots, C_k \text{ type-is } T \end{array}$$

A class definition introduces just necessary conditions regarding the type of the class – this is the standard case in object-oriented data models – while views are defined by means of both necessary and sufficient conditions. The symbol T denotes a type expression built according to the following syntax:

$$\begin{array}{ll} T \rightarrow C \mid & \\ \text{Union } T_1, \dots, T_k \text{ End} \mid & \text{(union type)} \\ \text{Set-of } [m,n] T \mid & \text{(set type)} \\ \text{Record } A_1:T_1, \dots, A_k:T_k \text{ End} & \text{(record type)} \end{array}$$

where $C \in \mathcal{C}_{\mathcal{S}}$, $A_i \in \mathcal{A}_{\mathcal{S}}$, and $[m,n]$ denotes an optional cardinality constraint.



A schema version in \mathcal{S} is defined by the application of a sequence of schema changes to a preceding schema version. The schema change taxonomy is built by combining the model elements which are subject to change with the elementary modifications, add, drop and change, they undergo. In this paper only a basic set of elementary schema change operators will be introduced; it includes the standard ones found in the literature (e.g., [4]); however, it is not difficult to consider the complete set of operators with respect to the constructs of the data model:

Add-attribute C, A, T End
Drop-attribute C, A End
Change-attr-name C, A, A' End
Change-attr-type C, A, T' End
Add-class C, T End
Drop-class C End
Change-class-name C, C' End
Change-class-type C, T' End
Add-is-a C, C' End
Drop-is-a C, C' End

In a framework supporting schema versioning, a mechanism for defining version coordinates is required. Such coordinates will be used to reference distinct schema versions which can then be employed as interfaces for querying extant data or modified by means of schema changes. We require that different schema versions have different version coordinates. At present, we omit the definition of a schema version coordinate mechanism and simply reference distinct schema versions by means of different subscripts. As a matter of fact, this approach is quite general in order to identify different versions. Any kind of versioning dimension usually considered in the literature could actually be employed – such as transaction time, valid time and symbolic labels – provided that a suitable mapping between version coordinates and index values is defined.

An evolving object-oriented schema is a tuple $\mathcal{S} = (\mathcal{C}_\mathcal{S}, \mathcal{A}_\mathcal{S}, \mathcal{SV}_0, \mathcal{M}_\mathcal{S})$, where:

- $\mathcal{C}_\mathcal{S}$ and $\mathcal{A}_\mathcal{S}$ are finite sets of class and attribute names, respectively;
- \mathcal{SV}_0 is the initial schema version, which includes class and view definitions for some $C \in \mathcal{C}_\mathcal{S}$;
- $\mathcal{M}_\mathcal{S}$ is a set of modifications \mathcal{M}_{ij} , where i, j denote a pair of version coordinates. Each modification is a finite sequence of elementary schema changes.

The set $\mathcal{M}_\mathcal{S}$ induces a partial order \mathcal{SV} over a finite and discrete set of schema versions with minimal element \mathcal{SV}_0 . Hence \mathcal{SV}_0 precedes every other schema version and the schema version \mathcal{SV}_j represents the outcome of the application of \mathcal{M}_{ij} to \mathcal{SV}_i . \mathcal{S} is called *elementary* if every \mathcal{M}_{ij} in $\mathcal{M}_\mathcal{S}$ contains only one elementary modification, and every schema version \mathcal{SV}_i has at most one immediate predecessor. In the following we will consider only elementary evolving schemata.

Let us now introduce the meaning of an evolving object-oriented schema \mathcal{S} . Informally, the semantics is given by assigning to each schema version a possible

legal database state – i.e., a legal instance of the schema version – conforming to the constraints imposed by the sequence of schema changes starting from the initial schema version.

Formally, an instance \mathcal{I} of \mathcal{S} is a tuple $\mathcal{I} = (\mathcal{O}^{\mathcal{I}}, \rho^{\mathcal{I}}, (\mathcal{I}_0, \dots, \mathcal{I}_n))$, consisting of a finite set $\mathcal{O}^{\mathcal{I}}$ of object identifiers, a function $\rho^{\mathcal{I}} : \mathcal{O}^{\mathcal{I}} \mapsto \mathcal{V}_{\mathcal{O}^{\mathcal{I}}}$ giving a value to object identifiers, and a sequence of version instances \mathcal{I}_i , one for each schema version \mathcal{SV}_i in \mathcal{S} . The set $\mathcal{V}_{\mathcal{O}^{\mathcal{I}}}$ of values is defined by induction as the smallest set including the union of $\mathcal{O}^{\mathcal{I}}$ with all possible “sets” of values and with all possible “records” of values. Although the set $\mathcal{V}_{\mathcal{O}^{\mathcal{I}}}$ is infinite, we consider for an instance \mathcal{I} the finite set $\mathcal{V}_{\mathcal{I}}$ of *active values*, which is the subset of $\mathcal{V}_{\mathcal{O}^{\mathcal{I}}}$ formed by the union of $\mathcal{O}^{\mathcal{I}}$ and the set of values assigned by $\rho^{\mathcal{I}}$ ([8]).

A version instance $\mathcal{I}_i = (\pi^{\mathcal{I}_i}, \cdot^{\mathcal{I}_i})$ consists of a total function $\pi^{\mathcal{I}_i} : \mathcal{C}_{\mathcal{S}} \mapsto 2^{\mathcal{O}^{\mathcal{I}}}$, giving the set of object identifiers in the extension of each class $C \in \mathcal{C}_{\mathcal{S}}$ for that version, and of a function $\cdot^{\mathcal{I}_i}$ (the *interpretation* function) mapping type expressions to sets of values, such that the following is satisfied:

$$\begin{aligned}
 C^{\mathcal{I}_i} &= \pi^{\mathcal{I}_i}(C) \\
 (\text{Union } T_1, \dots, T_k \text{ End})^{\mathcal{I}_i} &= T_1^{\mathcal{I}_i} \cup \dots \cup T_k^{\mathcal{I}_i} \\
 (\text{Set-of } [m, n] \ T)^{\mathcal{I}_i} &= \{\{v_1, \dots, v_k\} \mid m \leq k \leq n, v_j \in T^{\mathcal{I}_i}, \\
 &\quad \text{for } j \in \{1, \dots, k\}\} \\
 (\text{Record } A_1:T_1, \dots, A_h:T_h \text{ End})^{\mathcal{I}_i} &= \{[A_1 : v_1, \dots, A_h : v_h, \dots, A_k : v_k] \mid \\
 &\quad \text{for some } k \geq h, \\
 &\quad v_j \in T_j^{\mathcal{I}_i}, \text{ for } j \in \{1, \dots, h\}, \\
 &\quad v_j \in \mathcal{V}_{\mathcal{O}^{\mathcal{I}}}, \text{ for } j \in \{h+1, \dots, k\}\}
 \end{aligned}$$

where an open semantics for records is adopted (called **-interpretation* in [1]) in order to give the right semantics to inheritance. In a set constructor if the minimum or the maximum cardinalities are not explicitly specified, they are assumed to be zero and infinite, respectively.

A *legal* instance \mathcal{I} of a schema \mathcal{S} should satisfy the constraints imposed by the class definitions in the initial schema version and by the schema changes between schema versions. An instance \mathcal{I} of a schema \mathcal{S} is said to be legal if:

- for each class definition in \mathcal{SV}_0
Class C is-a C_1, \dots, C_h disjoint C_{h+1}, \dots, C_k type-is T , it holds that:
 $C^{\mathcal{I}_0} \subseteq C_j^{\mathcal{I}_0}$ for each $j \in \{1, \dots, h\}$,
 $C^{\mathcal{I}_0} \cap C_j^{\mathcal{I}_0} = \emptyset$ for each $j \in \{h+1, \dots, k\}$,
 $\{\rho^{\mathcal{I}}(o) \mid o \in \pi^{\mathcal{I}_0}(C)\} \subseteq T^{\mathcal{I}_0}$;
- for each view definition in \mathcal{SV}_0
View-class C is-a C_1, \dots, C_h disjoint C_{h+1}, \dots, C_k type-is T , it holds that:
 $C^{\mathcal{I}_0} \subseteq C_j^{\mathcal{I}_0}$ for each $j \in \{1, \dots, h\}$,
 $C^{\mathcal{I}_0} \cap C_j^{\mathcal{I}_0} = \emptyset$ for each $j \in \{h+1, \dots, k\}$,
 $\{\rho^{\mathcal{I}}(o) \mid o \in \pi^{\mathcal{I}_0}(C)\} = T^{\mathcal{I}_0}$;

Table 1. Semantics of the schema changes.

<u>Add-attribute</u> $\mathbf{C}, \mathbf{A}, \mathbf{T}$	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\dots, \mathbf{A} : v, \dots] \wedge v \in \mathbf{T}^{\mathcal{I}_j}\},$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Drop-attribute</u> \mathbf{C}, \mathbf{A}	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \pi^{\mathcal{I}_j}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\dots, \mathbf{A} : v, \dots]\},$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Change-attr-name</u> $\mathbf{C}, \mathbf{A}, \mathbf{A}'$	$\pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\dots, \mathbf{A} : v, \dots]\} =$ $\pi^{\mathcal{I}_j}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\dots, \mathbf{A}' : v, \dots]\},$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Change-attr-type</u> $\mathbf{C}, \mathbf{A}, \mathbf{T}'$	$\pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\dots, \mathbf{A} : v, \dots] \wedge v \in \mathbf{T}'^{\mathcal{I}_j}\} =$ $\pi^{\mathcal{I}_j}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) = [\dots, \mathbf{A} : v, \dots]\},$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Add-class</u> \mathbf{C}, \mathbf{T}	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \emptyset, \quad \rho^{\mathcal{I}}(\pi^{\mathcal{I}_j}(\mathbf{C})) \subseteq \mathbf{T}^{\mathcal{I}_j},$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Drop-class</u> \mathbf{C}	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \emptyset, \quad \pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Change-class-name</u> \mathbf{C}, \mathbf{C}'	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \pi^{\mathcal{I}_j}(\mathbf{C}'), \quad \pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}, \mathbf{C}'$
<u>Change-class-type</u> \mathbf{C}, \mathbf{T}'	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \pi^{\mathcal{I}_i}(\mathbf{C}) \cap \{o \in \mathcal{O}^{\mathcal{I}} \mid \rho^{\mathcal{I}}(o) \in \mathbf{T}'^{\mathcal{I}_j}\},$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Add-is-a</u> \mathbf{C}, \mathbf{C}'	$\pi^{\mathcal{I}_j}(\mathbf{C}) = \pi^{\mathcal{I}_i}(\mathbf{C}) \cap \pi^{\mathcal{I}_i}(\mathbf{C}'),$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$
<u>Drop-is-a</u> \mathbf{C}, \mathbf{C}'	$\pi^{\mathcal{I}_i}(\mathbf{C}) = \pi^{\mathcal{I}_j}(\mathbf{C}) \cap \pi^{\mathcal{I}_j}(\mathbf{C}'),$ $\pi^{\mathcal{I}_i}(D) = \pi^{\mathcal{I}_j}(D) \text{ for all } D \neq \mathbf{C}$

- for each schema change \mathcal{M}_{ij} in \mathcal{M} , the version instances \mathcal{I}_i and \mathcal{I}_j satisfy the equations of the corresponding schema change type at the right hand side of Tab. 1.

3 Using the Data Model

According to the semantic definitions given in the previous section, several reasoning problems can be introduced, in order to support the design and the management of an evolving schema:

- Global/Local Schema Consistency: an evolving schema \mathcal{S} is globally consistent if it admits a legal instance; a schema version \mathcal{SV}_i of \mathcal{S} is locally

- consistent if the evolving schema $\mathcal{S}_{\downarrow i}$ — obtained from \mathcal{S} by reducing the set of modifications $\mathcal{M}_{\mathcal{S}_{\downarrow i}}$ to the linear sequence of schema changes in $\mathcal{M}_{\mathcal{S}}$ which led to the version \mathcal{SV}_i from \mathcal{SV}_0 — admits a legal instance. In the following, a global reasoning problem refers to \mathcal{S} , while a local one refers to $\mathcal{S}_{\downarrow i}$.
- b. Global/Local Class Consistency: a class C is globally inconsistent if for every legal instance \mathcal{I} of \mathcal{S} and for every version \mathcal{SV}_i its extension is empty, i.e., $\forall i. \pi^{\mathcal{I}_i}(C) = \emptyset$; a class C is locally inconsistent in the version \mathcal{SV}_i if for every legal instance \mathcal{I} of $\mathcal{S}_{\downarrow i}$ its extension is empty, i.e., $\pi^{\mathcal{I}_i}(C) = \emptyset$.
 - c. Global/Local Class Disjointness: two classes C, D are globally disjoint if for every legal instance \mathcal{I} of \mathcal{S} and for every version \mathcal{SV}_i their extensions are disjoint, i.e., $\forall i. \pi^{\mathcal{I}_i}(C) \cap \pi^{\mathcal{I}_i}(D) = \emptyset$; two classes C, D are locally disjoint in the version \mathcal{SV}_i if for every legal instance \mathcal{I} of $\mathcal{S}_{\downarrow i}$ their extensions are disjoint, i.e., $\pi^{\mathcal{I}_i}(C) \cap \pi^{\mathcal{I}_i}(D) = \emptyset$.
 - d. Global/Local Class Subsumption: a class D globally subsumes a class C if for every legal instance \mathcal{I} of \mathcal{S} and for every version \mathcal{SV}_i the extension of C is included in the extension of D , i.e., $\forall i. \pi^{\mathcal{I}_i}(C) \subseteq \pi^{\mathcal{I}_i}(D)$; a class D locally subsumes a class C in the version \mathcal{SV}_i if for every legal instance \mathcal{I} of $\mathcal{S}_{\downarrow i}$ the extension of C is included in the extension of D , i.e., $\pi^{\mathcal{I}_i}(C) \subseteq \pi^{\mathcal{I}_i}(D)$.
 - e. Global/Local Class Equivalence: two classes C, D are globally/locally equivalent if C globally/locally subsumes D and viceversa.

□

Please note that the classical *subtyping* problem – i.e., finding the explicit representation of the partial order induced on a set of type expressions by the containment between their extensions – is a special case of class subsumption, if we restrict our attention to view definitions.

As to the *change propagation* task, which is one of the fundamental task addressed in the literature (see Sec. 4), it is usually dealt with by populating the classes in the new version with the result of queries over the previous version. The same applies for our framework: a language for the specification of views can be defined for specifying how to populate classes in a version from the previous data. Formally, we require a query language for expressing views providing a mechanism for explicit creation of object identifiers. At present, our approach includes one single data pool and a set of version instances which can be thought as views over the data pool. Therefore we consider update as a *schema augmentation* problem in the sense of [19], where the original logical schema is augmented and the new data may refer to the input data. The result of applying any view to a source data pool may involve OIDs from the source besides the new required OIDs to be created. The association between the source OIDs and the target ones should not be destroyed, and only the target data pool will be retained. In Section 4 an alternative approach will be discussed. Of course, at this point the problem of global consistency of an evolving schema \mathcal{S} becomes more complex, since it involves the additional constraints defined by the data conversions: an instance would therefore be legal if it satisfies not only the constraints of its the definition, but also the constraints specified by the views. Obviously, a schema \mathcal{S} involving a schema change for which the corresponding semantics expressed by the equation in Tab. 1 and the associated data conversions are incompat-

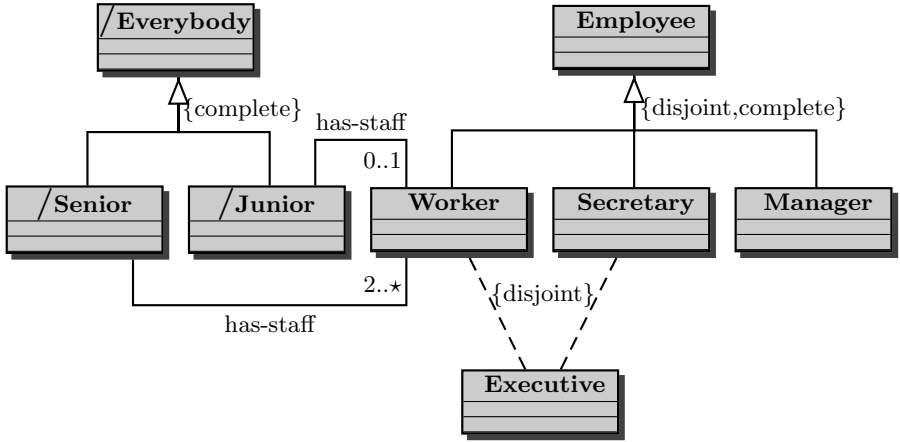


Fig. 1. The Employee initial schema version in UML notation.

ible would never admit a legal instance. In general, the introduction of data conversion views makes all the reasoning problems defined above more complex.

We will try to explain the application of the reasoning problems through an example. Let us consider an evolving schema \mathcal{S} describing the employees of a company. The schema includes an initial schema version \mathcal{SV}_0 defined as follows:

```

Class Employee type-is Union Manager, Secretary, Worker End;
Class Manager is-a Employee disjoint Secretary, Worker ;
Class Secretary is-a Employee disjoint Worker ;
Class Worker is-a Employee;
View-class Senior type-is Record has_staff: Set-of [2,n] Worker End;
View-class Junior type-is Record has_staff: Set-of [0,1] Worker End;
Class Executive disjoint Secretary, Worker;
View-class Everybody type-is Union Senior, Junior End End;

```

Figure 1 shows the UML-like representation induced by the initial schema \mathcal{SV}_0 ; note that classes with names prefixed by a slash represent the views. The evolving schema \mathcal{S} includes a set of schema modifications $\mathcal{M}_{\mathcal{S}}$ defined as follows:

```

( $\mathcal{M}_{01}$ )      Add-is-a Secretary, Manager End;
( $\mathcal{M}_{02}$ )      Add-is-a Everybody, Manager End;
( $\mathcal{M}_{23}$ )      Add-is-a Everybody, Secretary End;
( $\mathcal{M}_{04}$ )      Add-is-a Executive, Employee End;
( $\mathcal{M}_{45}$ )      Add-attribute Manager, IdNum, Number End;
( $\mathcal{M}_{56}$ )      Change-attr-type Manager, IdNum, Integer End;
( $\mathcal{M}_{67}$ )      Change-attr-type Manager, IdNum, String End;
( $\mathcal{M}_{68}$ )      Drop-class Employee End;

```

Let us analyse the effect of each schema change \mathcal{M}_{ij} by considering the schema version \mathcal{SV}_j it produces.

First of all, it can be noticed that in \mathcal{SV}_0 the **Junior** and **Senior** classes are disjoint classes and that **Everybody** contains all the possible instances of the record type. In fact, **Everybody** is defined as the union of view classes which are complementary with respect to the record type: any possible record instance is the value of an object belonging either to **Senior** or **Junior**.

Secretary is inconsistent in \mathcal{SV}_1 since **Secretary** and **Manager** are disjoint: its extension is included in the **Manager** extension only if it is empty (for each version instance \mathcal{I}_1 , $\text{Secretary}^{\mathcal{I}_1} = \emptyset$). Therefore, **Secretary** is *locally inconsistent*, as it is inconsistent in \mathcal{SV}_1 but not in \mathcal{SV}_0 .

The schema version \mathcal{SV}_3 is inconsistent because **Secretary** and **Manager**, which are both superclasses of **Everybody**, are disjoint and the intersection of their extensions is empty: no version instance \mathcal{I}_3 exists such that $\text{Everybody}^{\mathcal{I}_3} \subseteq \emptyset$. It follows that \mathcal{S} is locally inconsistent with respect to \mathcal{SV}_3 and, thus, globally inconsistent (although is locally consistent wrt the other schema versions).

In \mathcal{SV}_4 , it can be derived that **Executive** is locally subsumed by **Manager**, since it is a subclass of **Employee** disjoint from **Secretary** and **Worker** (**Manager**, **Secretary** and **Worker** are a partition of **Employee**).

The schema version \mathcal{SV}_5 exemplifies a case of attribute inheritance. The attribute **IdNum** which has been added to the **Manager** class is inherited by the **Executive** class. This means that *every* legal instance of \mathcal{S} should be such that every instance of **Executive** in \mathcal{SV}_5 has an attribute **IdNum** of type **Number**, i.e., $\text{Executive}^{\mathcal{I}_5} \subseteq \{o \mid \rho^{\mathcal{I}}(o) = [\dots, \text{IdNum} \mapsto v, \dots] \wedge v \in \text{Number}^{\mathcal{I}_5}\}$. Of course, there is no restriction on the way classes are related via subsumption, and multiple inheritance is allowed as soon as it does not generate an inconsistency.

The Change-attr-type elementary schema change allows for the modification of the type of an attribute with the proviso that the new type is not incompatible with the old one, like in \mathcal{M}_{56} . In fact, the semantics of elementary schema changes as defined in Tab. 1 is based on the assumption that the updated view should coexist with the starting data, since we are in the context of update as *schema augmentation*. If an object changes its value, then its object identifier should change, too. Notice that, for this reason, \mathcal{M}_{67} leads to an inconsistent version if **Number** and **String** are defined to be non-empty disjoint classes. Since the only elementary change that can refer to *new* objects is Add-class, in order to specify a schema change involving a restructuring of the data and the creation of new objects – like in the case of the change of the type of an attribute with an incompatible new type – a sequence of Drop-class and Add-class should be specified, together with a data conversion view specifying how the data is converted from one version to the other.

The deletion of the class **Employee** in \mathcal{SV}_8 does not cause any inconsistency in the resulting schema version. In \mathcal{SV}_8 the **Employee** extension is empty and the former **Employee** subclasses continue to exist (with the constraint that their extensions are subsets of the extension of **Employee** in \mathcal{SV}_6). Notice that, in a classical object model where the class hierarchy is explicitly based on a DAG, the deletion of a non-isolated class would require a restructuring of the DAG itself (e.g. to get rid of dangling edges).

3.1 Computational Properties of Reasoning

In this Section we only summarise the main results on the computational cost of reasoning in the proposed framework. □

Theorem 1. *Given an evolving schema \mathcal{S} , the reasoning problems defined in the previous Section are all decidable in EXPTIME with a □ PSPACE lower bound. The reasoning problems can be reduced to corresponding satisfiability problems in the \mathcal{ALCQI} Description Logic.*

This has been proved in [13] by establishing a relationship between the proposed model for evolving schemata and the \mathcal{ALCQI} Description Logic; for a full account of \mathcal{ALCQI} , see, e.g., [7]. To this end, a correct and complete encoding from an evolving □ schema into an \mathcal{ALCQI} knowledge base Σ has been provided, such that the reason□g problems mentioned in the previous section can be reduced to corresponding Description Logics satisfiability problems, for which extensive theories and well founded and efficient implemented systems exist. In particular, the semantics of any applied schema change $\mathcal{M}_{ij} \in \mathcal{M}_{\mathcal{S}}$ (which gives rise to an inclusion dependency between database instances according to Tab. 1) is translated into a corresponding *axiom* to be added to the knowledge base (see [13]). The encoding is grounded on the fact that there is a provable correspondence between the models of the knowledge base and the legal instances of the evolving schema.

Please note that the worst case complexity between PSPACE and EXPTIME does not imply bad practical computational behaviour in the real cases: in fact, a preliminary experimentation with the Description Logic system FaCT [18,17] shows that reasoning problems in realistic scenarios of evolving schemata are solved very efficiently.

As a final remark, it should be noted that the high expressiveness of the Description Logic constructs can capture an extended version of the presented object-oriented model, at no extra cost with respect to the computational complexity, since the target Description Logic in which the problem is encoded does not change. This includes not only taxonomic relationships, but also arbitrary boolean constructs, inverse attributes, n-ary relationships, and a large class of integrity constraints expressed by means of \mathcal{ALCQI} inclusion dependencies [8]. The last point suggests that axioms modeling schema changes can be freely combined in order to transform a schema in a new one. Some combination can be defined at database level by introducing new non-elementary primitives. □

4 Comparison with Other Approaches

The problems of schema evolution and schema versioning support have been extensively studied in relational and object-oriented database papers: [26] provides an excellent survey on the main issues concerned. The introduction of schema change facilities in a system involves the solution of two fundamental problems: the *semantics of change*, which refers to the effects of the change on the schema

itself, and the *change propagation*, which refers to the effects on the underlying data instances. The former problem involves the checking and maintenance of schema consistency after changes, whereas the latter involves the consistency of extant data with the modified schema.

In the object-oriented field (see [27,11] for the relational case), two main approaches were followed to ensure consistency in pursuing the “semantics of change” problem. The first approach is based on the adoption of *invariants* and *rules*, and has been used, for instance, in the ORION [4] and O₂ [12] systems. The second approach, which was proposed in [25], is based on the introduction of *axioms*. In the former approach, the invariants define the consistency of a schema, and definite rules must be followed to maintain the invariants satisfied after each schema change. Invariants and rules are strictly dependent on the underlying object model, as they refer to specific model elements. In the latter approach, a sound and complete set of axioms (provided with an inference mechanism) formalises the *dynamic schema evolution*, which is the actual management of schema changes in a system in operation. The approach is general enough to capture the behaviour of several different systems and, thus, is useful for their comparison in a unified framework. The compliance of the available primitive schema changes with the axioms automatically ensures schema consistency, without need for explicit checking, as incorrect schema versions cannot actually be generated.

For the “change propagation” problem, several solutions have been proposed and implemented in real systems [4,12,23,24]. In all cases, simple *default* mechanisms can be used or user-supplied conversion functions must be defined for non-trivial extant object updates.

As far as complex schema changes are concerned, [22] considered sequences of schema change primitives to make up high-level useful changes, solving the propagation to objects problem with simple schema integration techniques. However, with this approach, the consistency of the resulting database is not guaranteed nor checked. In [6], high-level primitives are defined as *well-ordered* sets of primitive schema changes. Consistency of the resulting schema is ensured by the use of invariants’ preserving elementary steps and by *ad-hoc* constraints imposed on their application order. In other words, consistency preservation is dependent on an accurate design of high-level schema changes and, thus, still relies on the database designer/administrator’s skills.

In this paper we have introduced an approach to schema versioning which considers a (conceptual) schema change as a (logical) schema augmentation, in the sense of [19]. In fact, the sequence of schema versions can be seen as an increasing set of constraints, as defined in Tab. 1; every elementary schema change introduces new constraints over a vocabulary augmented by the classes for the new version. An update of the schema is also reflected by the introduction of materialised views at the level of the data which specify how to populate the classes of the new version from the data of the previous version. Formally, in our approach the materialised views coexist together with the base data in the

same pool of data. In some sense, there is no proper evolution of the objects themselves, since the emphasis is given to the evolution of the schema.

More complex is the case when it is needed that a particular object maintains its identity over different version – i.e., the object evolves by varying its structural properties – and it is requested to have an overview of its evolution over the various versions. This is the case when a query – possibly over more than one conceptual schema – requires an answer about an object from more than one version.

In this case an explicit treatment of the partial order over the schema versions induced by the schema changes is required at the level of the semantics. Formally, this partial order defines some sort of “temporal structure” which leads us to consider the evolving data as a (formal) temporal database with a temporally extended conceptual data model [16,3,2]. With such an approach, different formal “timestamps” can be associated with different schema versions: all the objects connected with a schema version are assigned the same timestamp, such that each data pool represents a homogeneous state (snapshot) in the database evolution along the formal time axis¹. Objects belonging to different versions can be distinguished by means of the object’s OID and the timestamp. □

In such a framework, the (materialised) views expressing the data conversions can be expressed as temporal queries. In some sense, we can say that such a query language operates in a *schema translation* fashion[10] instead of a schema augmentation, where new data are presumed to be independent of the source data and an explicit mapping between them has to be maintained. Multischema queries can be seen as temporal queries involving in their formulation distinct (formal) timestamps. Moreover, in case (bi)temporal schema versioning is adopted, this “formal” temporal dimension has also interesting and non-trivial connections, which deserve further investigation, with the “real” temporal dimension(s) used for versioning.

Finally, the main application purpose of schema versioning is traditionally considered the reuse of legacy applications. Programs which were written and compiled in accordance to a schema version \mathcal{SV}_i are expected to still work even if the schema has been changed to \mathcal{SV}_j and the extant data have been changed accordingly: in a system supporting schema versioning, it would be sufficient to use the past schema version \mathcal{SV}_i to execute the application. In order to ensure full compatibility of current data with any past schema version (and applications using them), we have to introduce and enforce the notion of *monotonicity*. The schema modification \mathcal{M}_{ij} producing the schema version \mathcal{SV}_j from \mathcal{SV}_i is *monotonic* if the following inclusion relationship holds:

$$\overrightarrow{\mathcal{I}}_j \subseteq \overrightarrow{\mathcal{I}}_i, \text{ where } \overrightarrow{\mathcal{I}}_k = \{\mathcal{I}_k \mid \mathcal{I}_k \text{ is a legacy version instance for } \mathcal{SV}_k\}$$

Notice that not all the considered schema changes are monotonic: for example, the modification Change-attr-type $\mathbf{C}, \mathbf{A}, \mathbf{T}'$ is monotonic if and only if the new attribute type \mathbf{T}' is a subtype of the previous \mathbf{A} type. Furthermore, notice that,

¹ This case corresponds to the multi-pool solution for temporal schema versioning of snapshot data in the [11] taxonomy.

although a monotonic schema change implies a “reduction” in the current set of possible legal instances, the monotonicity constraint is not too restrictive in practice, as also useful “capacity-augmenting” changes can be considered monotonic: Add-class and Add-attribute (owing to the open record semantics) formally are. If all the schema changes in a sequence of modifications (e.g. $\mathcal{M}_{S_{1i}}$ which led from \mathcal{SV}_0 to \mathcal{SV}_i) are monotonic, the definition ensures that any legal instance of \mathcal{SV}_i was also a legal instance of \mathcal{SV}_0 . Therefore, any legacy query written for the schema \mathcal{SV}_0 can still be run on the current database instance connected with \mathcal{SV}_i , producing the same results as when \mathcal{SV}_0 was the current schema version. In case the sequence also contains non-monotonic changes, legacy queries are not ensured to still 100% properly work (of course they do if they do not involve the schema portion which underwent the non-monotonic change). The interesting issues connected with the monotonicity property and its enforcement will deserve a thorough investigation in our future research.

5 Conclusions and Future Developments

This paper deals with the support of database schema evolution and versioning by presenting and discussing a general framework based on a semantic approach, where the notion of change is seen as schema augmentation. As a consequence, we were able to define interesting reasoning tasks, to prove their computational complexity, and to reduce them to a reasoning problem in Description Logics for which inference tools do exist.

We are currently working to extend the framework presented in this paper to include a (simple) view language for data conversion in the schema augmentation context [19], for which the evaluation, consistency, and containment problems (under the constraints given by the evolving schema) could still be proved decidable. Once this view language is available, it would be possible to use it also for accessing the data through the schema versions, in the case when the schema evolves but a single database is maintained. Legacy applications could reuse the same query formulation related to a version of the schema different from the one modelling the actual data. This approach would also allow for multi-schema queries. In the database literature, the potentialities of queries involving multiple schema versions have been considered to a limited extent so far. For instance, relational queries [26] are usually solved with the help of a *constructed* schema, simply consisting of the union (or intersection) of all the attributes contained in the schema versions involved. Simple conversion functions are used to adapt data, stored according to a schema, to the constructed schema. On the other hand, this approach could be used as a basis for allowing the reformulation of multi-schema query answering as a *view-based query processing* problem, where powerful reasoning techniques on the query and the schemata can be deployed. In this way, complex relationships between extant data connected to different schema versions could be taken into account and sophisticated mechanisms could be used to combine them to construct the query answer in a provably correct way.

Further work will also be devoted to study the extensions/modifications of the proposed framework concerning the issues sketched in the previous Section.

References

1. S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1998. A first version appeared in SIGMOD’89.
2. A. Artale and E. Franconi. Schema integration of temporal databases. Technical report, University of Manchester, 1999.
3. A. Artale and E. Franconi. Temporal ER modeling with description logics. In *Proc. of Int’l Conference on Conceptual Modeling (ER)*. Springer-Verlag, November 1999.
4. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of ACM Int’l Conf. on Management of Data SIGMOD*, May 1987.
5. S. Bergamaschi and B. Nebel. Automatic Building and Validation of Multiple Inheritance Complex Object Database Schemata. *International Journal of Applied Intelligence*, 4(2):185–204, 1994.
6. P. Brèche. Advanced Principles of Changing Schema of Object Databases. In *Proc. of Int’l Conf. on Advanced Information Systems Engineering (CAiSE)*, May 1996.
7. D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi. Reasoning in expressive description logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2000. To appear.
8. D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modeling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 229–263. Kluwer, 1998.
9. D. Calvanese, M. Lenzerini, and D. Nardi. Unifying class-based representation formalisms. *Journal of Artificial Intelligence Research*, 11:199–240, 1999.
10. T.-P. Chang and R. Hull. Using witness generators to support bi-directional update between object-based databases. In *Proc. of ACM Int’l Symposium on Principles of Database Systems (PODS)*, 1995.
11. C. De Castro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.
12. F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and Database Evolution in the O₂ Object Database System. In *Proc. of Int’l Conf. on Very Large Databases (VLDB)*, September 1995.
13. E. Franconi, F. Grandi, and F. Mandreoli. A semantic approach for schema evolution and versioning in object-oriented databases. In *Proc. of Int’l Conf. on Rules and Objects in Databases (DOOD) as a stream of the First Int’l Conf. on Computational Logic (CL 2000)*. Springer-Verlag, July 2000.
14. F. Grandi and F. Mandreoli. ODMG Language Extensions for Generalized Schema Versioning Support. In *Proc. of ECDM’99 Workshop (in conj. with ER Int’l Conf.)*, November 1999.
15. F. Grandi, F. Mandreoli, and M. R. Scalas. A Generalized Modeling Framework for Schema Versioning Support. In *Proc. of Australasian Database Conf. (ADC)*, January 2000.
16. H. Gregersen and C. S. Jensen. Temporal Entity-Relationship Models - A Survey. *IEEE Transaction on Knowledge and Data Engineering*, 11(3):464–497, 1999.
17. I. Horrocks. FaCT and iFaCT. In *Proc. of Int’l Workshop on Description Logics (DL)*, 1999.

18. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In *Proc. of Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR)*, 1999.
19. R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proc. of Int'l Conf. on Very Large Databases (VLDB'90)*, 1990.
20. R. Hull and M. Yoshikawa. On the equivalence of database restructuring involving object identifiers. In *Proc. of ACM Int'l Symposium on Principles of Database Systems (PODS)*, 1991.
21. C. S. Jensen, J. Clifford, S. K. Gadia, P. Hayes, and S. Jajodia et al. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases - Research and Practice*, pages 367–405. Springer-Verlag, 1998.
22. S.-E. Lautemann. A Propagation Mechanism for Populated Schema Versions. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, April 1997.
23. S. Monk and I. Sommerville. A Model for Versioning of Classes in Object-Oriented Databases. In *Proc. of British Nat'l Conf. on Databases (BNCOD)*, July 1992.
24. D. J. Penney and J. Stein. Class Modification in the GemStone object-oriented DBMS. In *Proc. of Int'l Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, December 1987.
25. R. J. Peters and M. T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transaction on Database Systems*, 22(1):75–114, 1997.
26. J. F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1996.
27. J. F. Roddick and R. T. Snodgrass. Schema Versioning. In *The TSQL2 Temporal Query Language*. Kluwer, 1995.

Temporal Branching as a Conflict Management Technique

Roy Gelbard and Asher Gilmour

Information Systems Program
Department of Industrial Engineering and Management
Faculty of Engineering, Ben-Gurion University
Beer-Sheva 84105, ISRAEL
r_gelbard@yahoo.com ashergilmour@yahoo.co.uk

Abstract. The current research attempts to model a technique for managing conflicts within an integrated database. An integrated database being a database whose schema is an integration of several external database schemas, independent of each other. The independence of the systems allows conflicting values to be entered into the same data objects. For example, one system may hold the value of female for a specific data object where as another may hold male. The conflict is only discovered when the data objects are brought to the integrated database, and then there is need to resolve the conflict.

A technique to manage conflicts is developed based on version management, used in temporal databases, and the log-file approach used in more conventional technologies. The model combines temporal database tools with distributed database management tools. Thus it obtains greater flexibility than existing replication and log-file techniques and is more economic in record volume than the temporal approach.



1 Introduction



The architecture of distributed databases and the need for replicating data in the “Update Anywhere” architecture have made the appearance of data conflicts across databases inevitable [11]. This actually defeats the object of the database — to reduce conflicts within the organization. Because of this problem, conflict resolution mechanisms have been developed parallel to the update anywhere architecture [4,6,2]. The conflict resolution mechanisms are supposed to assure that the databases are free of internal contradictions, and that there is no recollection within them of conflicting data-sets that have been successfully resolved. The only way to trace conflicts is by using the recovery mechanisms that are based on checkpoints and rollback.

The result is that it is not possible to promptly retrieve data relating to situations at which conflicts occurred, as it is necessary to execute a preliminary rollback procedure. Even when a rollback is performed it is only possible to go back to a certain point of time, but not to examine data relating to sequence of time or a sequence of conflicts.



The current research suggests a model for conflict management. The proposed solution does not bury the conflict in the recovery mechanisms of a DBMS but keeps the conflict within the current data of the database. This model allows retrieval of data in light of different conflicts. The conflict management model is based on, but not identical to, the temporal oriented management of databases [12,17], which is adequate for dealing with a history of changes within a database.

The need to manage conflicts, and not only resolve them, exists in a variety of environments that include, among others, data management systems for clinical trials in the new drug application (NDA) process [14]. These systems are subject to strict audit oversight, described in the guidelines of the drug approval agencies (FDA in the United States and EUDRA in Europe [5,9,7]). In addition, these systems process thousands of records relating to the outcome of clinical trials. The data come from various separate systems, which can be described not only as distributed systems, but also as diverse systems where the data schemes of the different databases are not easily integrated. Moreover, even though the independence of the systems, it does not exempt them from the need to be free of conflicts and at the same time keep the original source records that caused the conflicts.

The characteristics of information systems used in the pharmaceutical research fields will lead, without doubt, to the development of solutions which are based on the use of a conflict dedicated log file [16].

The research examines and compares the log-file approach with the temporal database management approach.

2 The Problem

Environments that collect data from a variety of external systems are always subject to various conflicts in the integrated database created. The integrated database's data schema is an integration of several external database schemas, which are independent of each other. The independence of the systems allows the entering conflicting values to the same data objects. For example, one system can enter the value of female to a specific data object, whereas another system can enter male. The conflict is only discovered when the data objects are brought from the external systems to the integrated database, and then there is need to resolve the conflict.



The need to manage conflicts, and not only resolving them exists in many environments, among which is the new drug application (NDA) process. Before a company can introduce a new drug to a certain country, the local drug regulating authority must approve it. The NDA process is long and can take many years. During the process a vast amount of data is produced, and until currently has mainly been recorded on paper. Studebaker [14] describes the use of computers in the NDA process till 1992. The main conclusion is that computers are not used enough. The main uses of computers are to store the Case Report Forms (CRF) on electronic storage (by scanning) or to analyze the data, which are copied from the CRFs to the computer system. To our knowledge, the situation

has not changed much in recent past years. The regulatory organizations have realized the fact that there is need to use computerized systems for the NDA process, and that they even have quite a few advantages.

The FDA released a first draft guidance for the use of computers in clinical trials in 1997 [9], but still has not published the final ruling yet. In the ICH Guideline [7] there is a short reference to computerized data. They state that data must be reliable and that manipulated data must be comparable to its original data, this includes data imported from external systems. The acceptance of clinical trial outcomes, processed with the aid of computers for an NDA, depends highly on the compliance with the strict safety and reliability standards. When using electronic records for the entry of case CRFs, they are considered source records and the original values must be maintained within the data available for the final audit. The guidelines do not refer to conflicts and conflict resolution at all. There are only statements that data must be correct, and the various guidelines impose the use of audit trails within systems used for the trial. Although it is possible to satisfy the regulations by the use of a log-file (audit trail) and replication technologies, these solutions are very rigid, especially when retrieving data.

During the audit at the end of a trial, the auditors will want to see the whole evolution of various data objects, which with the existing technologies is a very rigorous task. The TB model gains much greater flexibility and additional features, which do not exist in other solutions.



3 Background Review

3.1 Distributed Data Management and Replication

In order to increase availability and concurrency of data in distributed data environments there is a need for data replication [4]. There are two major mechanisms for data replication: Multi-Master Replication and Snapshot Replication. The two mechanisms are also combined in hybrid configurations to meet a variety of needs [11]. Multi-master replication supports full table peer-to-peer replication, allowing master tables at all sites to be updated. Changes applied to any master table are propagated and applied directly to all other master tables, even in the event of a failure at a single master site. In snapshot replication updateable snapshots, which have smaller content than master tables, are updated without connection to what is going on in the underlying database and then propagated and applied to snapshot masters. Snapshots are refreshed from the master at time-based intervals or on demand. Any changes to the master table, since the last refresh, are then propagated and applied to the snapshot.

In order to maintain the concurrency of replicas at the various sites, many mechanisms have been developed. These include multi-version locking mechanisms such as the two-phase locking mechanism (2PL), optimistic and pessimistic concurrency control protocols and timestamp-ordering protocols [2,4,6].

It is not possible to assure locks at all sites, as some sites may be disconnected because of network failure or for other reasons. Because of this, it is not possible



to avoid conflicts in distributed systems based solely on replication methods and therefore they are combined with conflict resolution mechanisms.

Lock based protocols resolve conflicts, or at least prevent them, using transaction blocking or transaction abort. Transactions are blocked when requesting a data item in use by another transaction and depending on their priorities [6]. Timestamp ordering protocols use the transaction's timestamp to resolve conflicts; the transaction with a higher timestamp is awarded the block. Optimistic concurrency control uses the commit time to validate the transaction either by checking the data against the committed transaction, or checking data against existing running transactions.

To summarize this section it is worth mentioning that whatever replication method or concurrency control mechanism is chosen, conflicts are always inevitable, and the less synchronization there is between systems the greater the number of conflicts that occur.

3.2 Conflict and Conflict Resolutions



In distributed systems, and even in client-server systems, conflicts are great obstacles for the ongoing integrity of data within a database. Various types of operations can cause the conflicts; these go from simple typos to problems of replicating and synchronizing remote systems. Ensuring convergence in asynchronous replication environments is critical for nearly every application and is difficult on a large-scale basis [7]. But what happens if the same data object, e.g., the same column in the same row, is updated at two sites at the same time? This is known as an update conflict. To ensure convergence, update conflicts must be detected and resolved so that the data object has the same value at every site. Alternatively, update conflicts may be avoided by using the ownership limiting protocols described above.

Another type of conflict is a logical data conflict, where two or more data objects contradict each other. For example an electronic patient record (EPR) may have the blood type of a patient is A+, but in the request from blood bank a type B- blood transfusion is requested. These types of conflicts may be hard to detect and to resolve, as the conflict depends on the subject of the information systems involved. In order to minimize the occurrence of logical data conflicts, the information systems have to use specific logical checks integrated into the systems. The checks can be performed at the interface level or the database level. Though the problem of two users entering conflicting data from two different sites at the same time, or even at different times in asynchronous systems still exists. The resolution of these conflicts is not easy either, as it is not possible to know which data is the correct data. In the previous example, it could be that the EPR contains the wrong blood type (the patient is actually B- or another different blood type, such as AB-, A-, O+, etc.), or that a mistaken blood type was ordered from the blood bank (in this case, a very serious mistake).

In the model presented it is possible to manage all possible resolutions at the same time, until the true solution is decided upon. This makes it possible to run medical tools on the existing data, although there is an unresolved conflict.

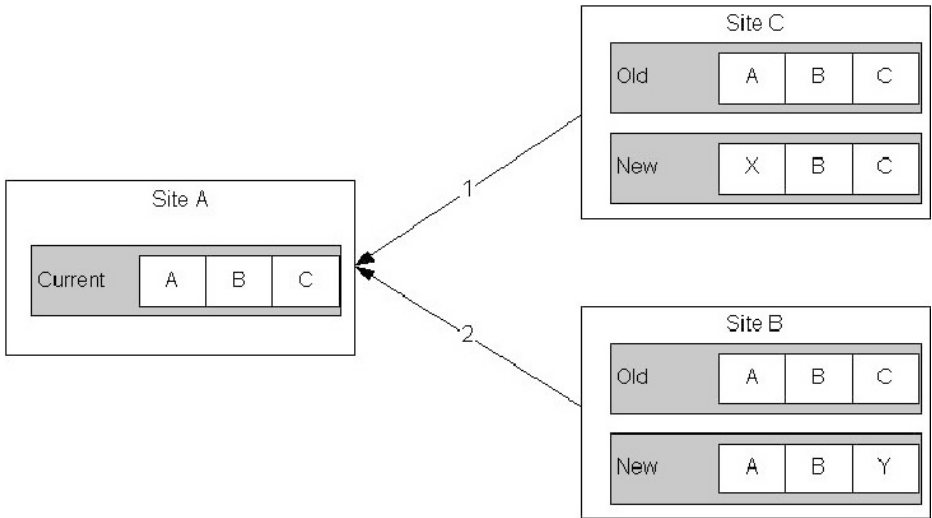


Fig. 1. Asynchronous Replication: Site B and C try to update site A with conflicting values.

The two types of conflicts occur as follows and are illustrated in Figure 1:

- At first all three sites have a record with field values A B C.
- Site B changes the record to A B Y and site C changes its local record to X B C.
- The two records are replicated to site A.
- The values from site B arrive at site A and as the old record at site B is identical to the record at site A, the replication is successful.
- The values from site C arrive at site A and as the old record at site C is now different from the record at site A (A B C against B C Y), a conflict arises and the replication fails. This is an update conflict.

If a mechanism exists that the procedure described in case 1 does succeed and the result is that the record at site A now reads X B Y, a logical data conflict may still arise. If, for example, the values X and Y cause a logical conflict (X = Blood type A+, Y = Blood type B-). It is important to state that for each conflict there may be a number of possible solutions. With all existing methods for conflict resolution there is no recollection of any of the alternative solutions after a conflict is resolved. Moreover, it is not possible to return to the point of time of a conflict and choose an alternative to the initial resolution and review the following data in light of the change. All this is possible in with the TB model.

3.3 The “Log File” Approach

In the FDA draft guidance [9] and the ICH guidelines for GCP [7] there is reference to the audit trail, or log-file. The log-file must record all changes to

the database, which record was changed, what field was changed, from what, to what, when and who performed the transaction. Auditing database data has generally meant keeping a separate record, similar to the one described above, or audit trail of selected changes made to the database [16]. For example, Oracle provides the capability to audit user, action, and date for access to selected object types but requires a user to write triggers to record changes to data values. While this straightforward mechanism does accomplish the task, its use for large and complex databases rapidly generates huge volumes of data that require sophisticated searching to identify particular changes of interest. A simple audit log of database changes is practical only if one hopes that it will never be needed. Audit logs are routinely needed in the pharmaceutical industry and will soon be a common requirement for other industries subject to regulatory oversight, such as software development processes subject to ISO validation. Searching through a huge audit log is not a reasonable way to answer an auditor's questions about the history of an object that may contain, or be associated with, hundreds of component objects.

The HP laboratory database solution described in [16] takes the audit trail slightly further by implementing it in an object oriented environment. The HP system keeps old data objects in such a way that they are easily retrieved. Although the system has a specific solution for the problem in hand, it is still only possible to create a single audit path and is not useful in the most common DBMS, the RDBMS.

3.4 Temporal Databases

As mentioned before, a temporal database can be seen as the extreme case of concurrency control where all old versions of data objects are kept [4]. There are various methods of maintaining temporal databases. Some models use a time instance added to each tuple to identify the version of the data object; others use intervals and even triple notations [4]. In a temporal database, current data and old data are managed perfectly symmetrically [13]. Although it is possible to add time-related fields to records in a standard relational database, it is very difficult to manage the temporal data effectively. As mentioned before, a great deal of temporal data have models and various derived query languages. When giving a general description of a temporal database, the temporal cube is presented (Figure 2).

The temporal cube replaces the standard relation with a temporal oriented relation. The cube is defined as the collection of all tables that relate to all points of time. The definition of time depends on the application and can be either transaction time or valid time [3,4,8,12,13]. The cube allows the accumulation of temporal oriented data for each data object. The data values are never replaced or deleted, data updates are expressed by the recording of a new value to the table that belongs to the time of the update. The value will then appear in all future tables that belong to the future times, until a new value is entered. It is clear that the cube model, in its simple form, has many duplicate values, though there are many temporal models that reduce the volume of that data. In order to

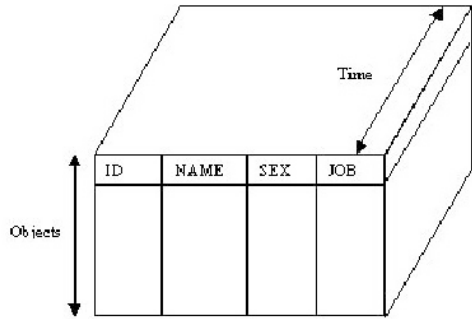


Fig. 2. The Temporal Cube: standard data object with the additional time dimension.

reduce the volume of data in the temporal data model it is possible to distinguish between constant attributes and variable attributes in a record. Constant values never change once entered, such as date of birth or sex, and therefore may be recorded only once. Variable attributes may change over time and therefore need to be treated differently. An interesting state is when a variable attribute ceases to exist in the database, the attribute is assigned a NULL in all future tables. When discussing temporal operations, the following operations are recognized; Time selection is the operation of choosing a vertical slice of the time cube. Time selection takes all the tables between two points in time. 'Some when' selection takes any data objects that satisfy the query anywhere in the data cube. By the use of 'Every when' selection, data objects are chosen so that they satisfy a query at all time points in the data cube. Temporal projection is virtually the same as a projection in an ordinary RDMS, only with the addition of the temporal dimension. The additional dimension does make the operation far more complicated. Finally there is the temporal join, which is yet again similar to an ordinary join, with the difference of the possibility of joining various time dimensions and give the option of comparing multiple time slices in one join.

In temporal and real-time databases there are the similar problems of conflicts and concurrency control. The same concurrency control mechanisms are used as with replication techniques, i.e. lock based methods, optimistic methods and timestamp ordering protocols [3,12]. A possible solution for conflict resolution is to create a new time slice for each conflict resolution in a temporal database, though this becomes very costly in data volume, as the whole data schema is duplicated.

3.5 Temporal Versioning

Many applications require databases that support both temporal data and version management. However most of proposed temporal database models support neither alternative versions nor schema versioning and version management do not have time aspects support. Schema versioning as used in standard temporal

database models is broken to schema evolution and schema versioning [8]. The definitions, in the latest consensus glossary, for these are:

- *Schema Evolution*: A database system supports schema evolution if it permits modification of the database schema without loss of data.
- *Schema Versioning*: A database accommodates schema version if it allows the querying of all data, both retrospectively and prospectively.

From this it is understood that if a database supports schema evolution, it does not necessary allow the use of the old data schema. In Lü et al. [10] a difference between data versioning and schema versioning is established. They introduce a new data concept “temporal versioning”, which provides a uniform way of understanding temporal data and versions. They present a data model called temporal versioning model (TVM), which incorporates temporal versioning semantics of the real world into the object-oriented database model. Their model supports multiple dimensional time to overcome the limitation of existing temporal database models that support one valid time attribute and one valid transaction time only. A temporal version can be accessed by its time information, or its identifier or values of properties. A change in the values of a data object creates a new data version. Within a schema there may be data multiple versions. In addition a change to the schema creates a new schema version. By definition each schema may have many versions. A new version with its own life span is created after each update. Each object (instance of class) may have many versions and each version has its own life span. If more than one version of an object is valid at a certain time, these versions may be distinguished by version- identifiers. The fact that a temporal version can be accessed by its time information, or its identifier or values of properties, helps to overcome the limitations of accessing temporal data only by its time information or by its version identifier.

4 The Temporal Branching Method

The temporal branching model extends the traditional relational data schema by giving the schema additional dimensions similar to temporal databases. In addition the TBM introduces a set of components supporting the detection and handling of conflicts. By the addition of these components it is possible to manage and hold parallel resolutions of data conflicts and keep various versions of data objects. In this section the components and the behavior of the Temporal Branching (TB) model are described, relating both to the components in the database and to the principles of managing data. The section about the model’s behavior describes the behavior during routine running of the database, and the instances that a conflict is discovered. The model can be implemented in any commercial relational database without the need for the addition of components to DBMS.

4.1 Additional Dimensions: Resolutions and Versions

Two dimensions are added to every data object in the database: the resolution dimension and the version dimension. In order to support these two dimensions each record has two added fields: the resolution number and version number fields. Every time a conflict is detected, it may be resolved with more than one solution, creating a new branch for each resolution along the resolution dimension. A set of data objects, each with one resolution, can be mutually defined as a version. When a version is set, there may not be any duplicate data objects by definition. In Figure 3 it is possible to see a conceptual overview of the TB model. On the uppermost level we have the base version, version 0, where all data objects have only a single resolution.

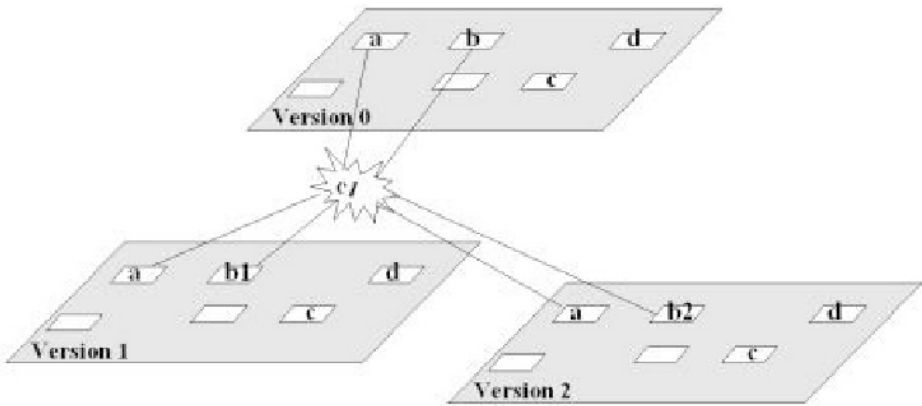


Fig. 3. A conceptual overview of the TB model. Resolutions are grouped to versions. Ungrouped resolutions are displayed together with the version in which the conflicts occurred.

At some point a conflict is detected ($c1$) and multiple resolutions are created, $b1$ and $b2$ (obviously it is possible to define more resolutions). Here by we have created new resolution dimensions for the data object b , these dimensions are marked with numbers, resolutions 1 and 2. The data objects may now be collected and set in new version, creating versions 1 and 2. Version 1 consists of resolution 1 from conflict $c1$ and version 2 is constructed from resolution 2 from conflict $c1$. It is obviously possible to create other versions with other combinations of resolutions, but this may be avoided, as other combinations may be meaningless to the application of the database. The version have a mapping between them, in this case version 0 will be the parent of versions 1 and 2. It is important to note that not all data objects are duplicated to the new versions. Data objects that were not involved in the conflicts that occurred between the versions (for example $C1$ that occurred between version 0 and version 1) are inherited from the parent version via the version tree.

4.2 The Model's Components

The model has the following physical components: Temporal log file, Record status field, Reference tables, Version checkpoints and Sub-version checkpoints.

Temporal Log File. In order keep track of all the changes the model uses a temporal log file, which is similar to the standard log file used in most databases. The log file consists of the following fields: Table Identifier, Record Identifier (the primary key, which is defined as a Sequence Number, unique within a table), Field Identifier, Old Value (optional, but recommended for enhanced control), New Value, Timestamp, User Identifier. It is possible for control reasons to use the reference tables in order to refer to the different database access permissions of a specific user.

The Record Status Field is added to every record. The status field is a Boolean field to avoid physical deletion of records, and is used to represent a logical deletion. The field denotes if a field is deleted or not. This is in addition to the dimension fields.

The Reference Tables will refer to the version code relevant to the data object. The reference table contains the following fields: Category Code (Optional), Category Item Code, Description of the Item, and Version identifier. The version identifier is part of the unique key of the referenced table.

Version Checkpoint (VCP) is the last point of time that is conflict free, before the declaration of a new branch. This means the last sub-VCP before the declaration.

Sub-version Checkpoint is a point in time that is conflict free and is marked as a label in the log file. The time interval that a sub-VCP is marked is adjustable to any length of time.

4.3 The Model's Behavior

The model recognizes two situations of operation: routine operation and conflict discovery. The model acts differently in both situations. It is possible to implement the behavior with the use of standard SQL statements (ANSI-SQL or any other compatible dialect of the different vendors), and there is no need for the development of customized operators. The use of stored procedures and triggers is particularly efficient here.

Routine Behavior. For the routine running of the model some simple procedures have been developed. The procedures are easily implemented using triggers and stored procedures in a relational database. The model works in a similar manner to standard DBMS where the transaction is written to a log prior to being committed to the database. This is in addition to the process of routinely adding checkpoints at any time that there are no locks, and in a fixed time interval.

I. Writing to the Temporal Log File

Each transaction is written in a temporal log-file and only later written to the database itself. A 'before insert' trigger triggers the write operation, which is implemented on all tables in the database.

II. Marking Sub-version Checkpoints

A fixed time interval is set in the system during which the system tries to mark a sub-VCP. The success of such an operation depends on the following two conditions:

- No locks in the database — a check done automatically by an RDBMS.
- No Conflicts — A check implemented by the analysis of current and future values during a write trigger (the 'before insert' trigger mentioned before). The trigger contains all the definitions of possible conflicts in the system including functional conflicts, which depend on the violation of functional constraints between various fields in the database.

Conflict Discovery Reacting Behavior. As described before, various mechanisms can uncover conflicts. For example, a write trigger in the temporal log-file can discover a conflict. The trigger runs comparison operations between current values and future values of different records that are supposed to fulfill defined functional relationships. On the discovery of a conflict the following operations are done:

I. Temporal Roll Back

The database is rolled back according to the data in the temporal log-file until the last sub-version checkpoint. I.e. all transactions are undone to the situation where the conflict occurred.

II. Freezing of Version's Configuration

The configuration of the database at the point of the last sub-version checkpoint, up to which the database is rolled back, is defined "the final configuration of version X", and is marked as a version checkpoint in the log-file.

III. Defining Possible Resolutions to the Conflict

An expert user (with understanding of the function of the system, such as a physician using a clinical database), and not a technical user (such as the DBA), inputs manually the possible resolutions to the conflict. It is also possible for some kind of expert system.

- Each verified resolution is characterized by fixing a set of new values in the fields in which the conflict was discovered.
- The system identifies each resolution by a version number, which is built as an identifier of a node in a tree.
- During each resolution the relevant records are duplicated to each of the new versions defined by the user.

Note that contrary to temporal databases in the current model, only the conflicting records are duplicated, and not the whole data schema. Even though new schemas are defined, the original data schema is used and there is no transformation to a new schema.

IV. Relevant Reference-Table's Records Duplication

The relevant records in the reference table are defined according to the nature of the conflict resolution for each version. If there are no changes in the reference table, it is possible to advance the version code (which is a part of the primary key of these tables). In principle it is possible to define a null value to denote compatibility of records to all the existing versions, or, a different value as compatibility identifier of records to higher versions in the version tree (i.e. to all the versions prior to the current node).

4.4 Version Management and Administration

As described above a human expert, or expert system, defines the various conflict resolutions. The definition of versions happens in a similar way as the definition of conflict resolutions. It is obvious that if the versions are not administered properly there are good chances of exponential explosion along the conflict resolution dimension. This would destroy the gain achieved by not duplicating all objects along the time dimension as in the temporal oriented approach. The model allows versions to be cancelled (but not deleted) at any time with the entry of more information to the database. The functional purpose of the versions is triple fold:

- I. To keep the database 'going' at the time of a data conflict.
- II. Avoid deleting data from the database that was not valid and causing conflicts. This is so that the original situation is not lost in the recovery mechanisms.
- III. To perform analysis of parallel options of values in data objects.

Therefore, there is no need to keep many parallel versions actions, and as in any other fast growing data environment needs to be under strict administration. With proper administration the volume of the database can be kept down.

5 Discussion and Conclusions

The current research presents a technique for managing conflicts within an integrated database, a database whose schema is an integration of several external database schemas, which are independent of each other. The independence of the systems allows the entering conflicting values to the same data objects. For example, one system can enter the value of female to a specific data object, whereas another system can enter male. The conflict is only discovered when the data objects are brought from the external systems to the integrated database, and then there is need to resolve the conflict.

The research introduces a technique to manage conflicts based on version management and log-file approach. Thus, the TB model combines the temporal versioning approach with the log-file technique and conflict resolution methods (used in database replication models). Due to this, the model obtains greater flexibility than existing replication and log-file techniques, and is more economic in record volume than the classic temporal oriented approach. The TB model constructs a 'time tree' similar to the versioning approach and different from the classic temporal oriented approach, which presents only a single temporal path. The time tree denoted not only the time dimension, but also the versions' dimensions. Each node within the tree presents a conflict and the number of branches denotes the number of possible solutions to the conflict that were continued. The time tree presents three different aspects of the model: Time dimension that presents the presidencies and chronological appearance of events. Conflict aspect that presents all the data leading to the conflict, and aspect of Possible resolutions to conflicts.

The time tree presented allows managing parallel solutions to conflicts without the need to drop any possible solution at any time. This is contrary to existing mechanisms, the log-file and replications, which allow only one single solution. Due to this character the TB method is especially useful in environments where there is need to manage more than one possible solution to a conflict. The need to manage more than one possible solution arises in systems where one wants to allow a spectrum of analysis operations. These operations include investigating past events and their consequences by scanning the time tree backwards. 'What if' analysis based on true experiences accumulated on parallel branches on the 'time tree', and the possibility of 'regretting transactions' by simply moving to a parallel branch instead of having to recovering the original state and starting all transactions over again. Standard log-file based systems and replication methods do not allow this kind of operation in a simple manner. With the other techniques it is possible to perform a rollback mechanism to return to an initial state of conflict, but is not possible to continue from that point using a different path from the one chosen at the time that the original conflict was resolved.

There are many systems that may take advantage of this unique character of the TB model, these include medical and business information systems. In the medical field the model is relevant to clinical trial data management systems, electronic patient record (EPR) systems and diagnostic systems which use the EPR systems as their knowledge base. These two fields may use this kind of operation in order to manage parallel solutions such as parallel diagnosis or hypothesis. Decision-makers can follow various diagnoses and hypotheses and see how they evolve over time with the addition of new data to system.

In addition to the functional capabilities of the model in the area of conflict resolutions, the model must be applicable in existing commercial database technologies. This is in order to stand up to the reliability standards of the regulatory organizations. There are two implications: one, the applicability of the TB schema by use of a relation schema, and two, the applicability of data retrieval,

for both conflict resolution and querying of data by the use of standard SQL statements.

The TB model allows keeping all relevant data in a time-tree and source data in one single integrated database, applicable in any commercial RDBMS. It is possible to use standard SQL statements to perform 'what if' queries. In addition the TB is economic in data volume relative to traditional temporal approaches, due to the fact that only the data relating to a conflict is duplicated, and not the whole data schema.

To Summarize. Although replication and log-file technologies do supply tools for conflict resolution, these tools lack flexibility and allow only a single solution to a conflict. The TB model, on the other hand, is completely flexible. The flexibility is obtained by the following capabilities:

- The possibility of creating multiple 'time dimensions' by splitting the time path in to unlimited branches, any time a conflict occurs.
- The possibility to manage the data in every branch at the same time, and perform 'what if' analysis for each possible solution of a conflict.
- Investigating data relating to the time a conflict occurred and after each possible solution, without performing any recovery operations such as rollback.

Contribution and Further study. The current research is relevant to a variety of fields such as: distributed databases, schema integration, data replication, conflict resolution and temporal database management. The research relates to information systems used in medicine and business. The research deals with medical systems such as clinical trial data management systems, EPR systems and medical expert systems that draw their data from underlying active medical databases. In the business field it is possible to perform analysis of costs of error such as the cost a specific re-occurring conflict within an organization. For example, it is possible ask in a package delivery system: "How much did the wrong resolution of package addressing and delivery cost the company during period X?".

The continuation of the research is being conducted at four levels: At a Technical level, where the model is being implemented on a commercial RDBMS. At an Experimental level, where the log-file approach will be compared with the TB using real data of clinical trails. At a Theoretical level, enriching the model, by creating additional SQL statements to support 'what if' analysis and retrieval of data across conflict. And, at a Practical level, exploring more implications for the TB model.

References


1. Ahamad, M., Ammar, M.H. and Cheung, S.Y., "Replicated Data Management in Distributed Systems". In: Readings in Distributed Computing Systems, Thomas L. Casavant and Mukesh Singhal (Eds.), IEEE Computer Society Press, 1994.

2. Anastassopoulos, P. and Dollimore, J., "A Unified Approach to Distributed Concurrency Control". In: *Readings in Distributed Computing Systems*, Thomas L. Casavant and Mukesh Singhal (Eds.), IEEE Computer Society Press, 1994.
3. Combi, C. and Shahar, Y., "Temporal Reasoning and Temporal Data Maintenance in Medicine: Issues and Challenges", *Computers in Biology and Medicine*, Vol. 27, No. 5, 1997, pp. 353-368.
4. Elmasri and Navathe, *Fundamentals of Database Systems*, Benjamin Cummings, USA, 1994.
5. FDA, "21 CFR Part 11. Electronic Records; Electronic Signatures; Final Rule", *Federal Register*, Vol. 62, NO. 52, 1997, pp. 13430-13466.
6. Hong, S. H. and Kim, M. H., "Resolving Data conflicts with multiple versions and precedence relationships in real-time databases", *Information Processing Letters*, Vol. 61, Feb. 1997, pp. 149-156.
7. ICH, "ICH topic 6 — Guideline for Good Clinical Practice", The European Agency for the Evaluation of Medical Products (EUDRA), 1996.
8. Jensen, Christian S. and Dyreson, Curtis E. (Eds.), "The Consensus Glossary of Temporal Database Concepts — February 1998 Version", *Temporal Databases — Research and Practice, Lecture Notes in Computer Science 1388*, Springer-Verlag, Berlin, 1998, pp. 368-405.
9. Lepay, David A., "Guidance for Industry. Computerized Systems Used in Clinical Trials", (draft), *Federal Register*, Vol. 62, 33094, 1997.
10. Lü Jiang, Barclay, P. and Kennedy, J., "On temporal versioning in temporal databases", *Informationssystem-Architekturen*, Vol. 3, Iss. 1, Sept 1996, pp. 38-40.
11. Oracle, "Introduction to Oracle7 Advanced Replication", White Paper, www.oracle.com, October 1996.
12. Özsoyogulu, Gultekin and Snodgrass, Richard T., "Temporal and Real-Time Databases: A Survey", *IEEE Trans. on Knowledge and Data Engineering*, vol. 7, No. 4, August 1995, pp. 513-532.
13. Shiftan, Y., "Managing Table Databases Incorporating the Time Dimension", (Hebrew), *Computers*, Sept. 1990, Israel.
14. Studebaker Joel F., "Computers in the New Drug Application Process", *J. Chem. Inf. Comput. Sci.*, 1993 (33), pp. 86-94.
15. Tansel, Clifford, Gadia, Jajodia, Segev and Snodgrass, *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, USA, 1993.
16. Timothy P. Loomis, "Audit History and Time-Slice Archiving an Object DBMS for Laboratory Databases", *HP Journal*, Article 10, August 1997.
17. Yu Wu, Sushil Jajodia and Sean Wang, X., "Temporal Database Bibliography Update", *Temporal Databases — Research and Practice, Lecture Notes in Computer Science 1388*, Springer-Verlag, Berlin, 1998, pp. 368-405.

Evolving Relations


Ole G. Jensen and Michael H. Böhlen*

Department of Computer Science, Aalborg University,
Frederik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark,
<guttorm|boehlen>@cs.auc.dk

Abstract. This paper presents a framework for evolving relation schemas that is based on *conditional schema changes* and *tuple versioning*. With each tuple a *recorded schema* and a *conceptual schema* is associated. This allows for a simple and semantically clean solution to the problem of schema mismatches that arise when the schema of a database is changed and some data no longer fits the schema. Specifically, no data needs to be migrated to the new schema, and no special null values are required. We precisely define *evolving schemas* in terms of *schema segments* and corresponding *attribute mappings*, present an algorithm to compute answers to queries over evolving schemas, and prove that the query answers consider the maximal set of schema segments consistent with the evolving schema. 

1 Introduction

Databases are frequently modified and many modifications result in changes to the database structure [22]. In stark contrast, applying schema changes to a populated database is still an open issue. The main difficulty is that after a schema change some data no longer fits the schema. Resolving this mismatch is a notorious problem, and a semantically clean and simple solution has not yet emerged. This paper presents a formal and intuitive solution that is both faithful to the schema change and the stored data.

We propose a solution where *each tuple* is associated with a conceptual and a recorded schema, respectively. The *conceptual schema* denotes the logical schema of a tuple, i.e., the schema a tuple is supposed to have. The *recorded schema* denotes the actual schema of the stored tuple. Assume an employee relation with schema $\{N_{\text{ame}}, C_{\text{ontinent}}, U_{\text{nit}}\}$ and a tuple $t = (LiChen, Asia, db)$. The recorded and the conceptual schema of t is $\{N, C, U\}$. A schema change that adds a G_{roup} attribute to the employee relation changes  the conceptual schema of t to $\{N, C, U, G\}$. The recorded schema of t is left unchanged. The accurate modeling of conceptual and recorded schemas allows to selectively and precisely resolve potential mismatches between the two.

The separate handling of recorded and conceptual schemas for each tuple is termed *tuple versioning*. Tuple versioning uses a finer granularity than traditional schema versioning [9]. On the one hand, this avoids the problems of

* This research was supported in part by the Danish Technical Research Council through grant 9700780 and Nykredit, Inc.

data migration where data has to be migrated to a new schema in response to schema changes, and multiple NULL values have to be used to distinct unknown attribute values from missing attributes. On the other hand, tuple versioning naturally supports *conditional schema changes*, i.e., schema changes that shall be applied to a selected subset of the extension of a relation. For example, subdividing the database unit into groups is a conditional schema change that adds a G_{group} attribute to the employees in the database unit. Conditional schema changes are the most general type of schema changes, and properly subsume both unconditional changes and changes along the time dimension.

We formalize how to generalize a (static) relation to an *evolving relation*. Evolving relations consist of evolving schemas and evolving instances. The possibility to let schemas and instances evolve asynchronously allows us to be faithful to the schema change *and* the stored data. An *evolving schema* consists of a set of schema segments and corresponding attribute mappings. *Attribute mappings* maximize the potential to treat different segments uniformly. For example, adding new syntactic constructs to the query language to explicitly identify specific segments is not necessary if attribute mappings are present. A *schema segment* associates a schema and a qualifier. The *qualifier* derives from the condition of the conditional schema changes and identifies the tuples associated with the respective schema.

When querying evolving relations, a query usually does not apply to all segments. For example, a query that asks for the G_{group} attribute does not apply to segments with schemas that do not include a G_{group} attribute and do not have attribute mappings to derive the G_{group} attribute. We present an algorithm to compute queries over evolving relations. The MAP algorithm maps a set of source attributes to a target attribute. This can be used to link the attributes used in a query to the attributes of a schema segment. The TRANSFORM algorithm uses MAP to rewrite a query to a specific segment of an evolving relation. We prove that the set of segments a query can be rewritten to is maximal and that a conditional schema change does not reduce this set.

In Section 2, we present requirements to evolving relation schemas and summarize our solution. Section 3 formally defines evolving relations. Section 4 presents conditional schema changes, which can be described in terms of three conditional schema change primitives. We illustrate how to use conditional schema changes to express a wide range of schema change operations proposed in the literature. In Section 5, queries over evolving relations are investigated. We show that tuple versioning allows to accurately answer queries and give algorithms to compute such answers. Related work is given in Section 6, and Section 7 presents conclusions and directions for future work.

2 Requirements for Evolving Relation Schemas

This section presents three requirements for evolving relation schemas. The focus is on general requirements from which more specific ones can be derived. With each requirement we discuss its consequences, and sketch some of the more specialized requirements that can be derived from it.

R1—Selective schema changes. Conceptually, a schema change is a change of the properties of a set of tuples. Assume the relation schema $E_{\text{employee}}(\text{Name}, \text{Continent}, \text{Unit})$. Each employee tuple has a set of properties that correspond to the attributes in the relation schema. The properties of an employee are: his/her name, the home continent, and the unit s/he works in.

To illustrate selective changes assume a schema change that adds a G_{group} attribute to the E_{employee} relation. This modification extends the properties of all employees. Often it is natural that only some employees change their properties. For example, large units shall be sub-divided into groups, whereas no additional division shall be imposed on small units. A schema change is *selective* if it applies to a subset of the extension of a relation. For example, a schema change that adds a G_{group} to the employees in the database unit is selective, since it does not change the properties of the employees in, e.g., the information systems unit. Note that selective schema changes properly subsume (universal) schema changes. It is always possible to have the selection choose the entire extent of a relation.

The example also illustrates that it is natural to have selection criteria that are not based on the time. Because evolution over time is very common, schema changes have often been investigated in the context of temporal databases [8, 2, 21, 6]. Our approach is more general in the sense that the selection of the tuples can be based on any attribute of the relation. In our example, the unit an employee is working in is used as a selection criteria.

A consequence of requirement R1 is that a relation may become heterogeneous. Typically, the tuples in an evolving relation will no longer have the same schema. The tuples are still strongly related, though and we often want to access them uniformly. Therefore, it is important to logically group them together in an evolving relation.

R2—Transparent schema changes. The next requirement ensures that schema changes do not enforce a change of how users interact with the database. Specifically, legacy queries and database updates shall remain valid when the schema is changed. For example, consider the E_{employee} schema, $E(N, C, U)$, and assume a schema change that splits N_{name} into F_{first} and L_{last} names for employees from the European Union. A transparent schema change ensures that legacy applications can still specify N_{name} when adding tuples to the database.

A direct consequence of this requirement is that users do not have to be aware of the individual schema segments when specifying schema changes. The schema change remains the same whether applied to a static schema or an evolving schema with multiple segments. Thus, a user is neither bothered by nor aware of the fact that the schema is evolving. This also holds true for queries. Assume the above schema change and a query $Q_1 = \pi[N]\sigma[U = db](E)$, that asks for the names of employees in the database unit. A transparent schema change guarantees that query Q_1 remains valid without additional information. Specifically, the application does not have to be changed because the name was split into first and last name.

R3—No value-encoded schema information. When the schema of a populated database is modified some data no longer fits the schema. Typically, such mis-

matches are resolved by migrating non-fitting tuples to the new schema using some kind of null values for missing attributes. Consider the E_{employee} relation with schema $E(N, C, U)$ and a schema change that sub-divides the database unit into groups. This leads to the new schema $E'(N, C, U, G)$. Since the tuples in E do not have an associated G_{group} attribute, they do not fit into E' . As illustrated in Table 1, they can be forced into the new relation schema if, e.g., null values are substituted for the missing attributes [18].

Table 1. Example Instances of E_{employee}

E				E'				
	Name	Continent	U_{unit}		Name	Continent	U_{unit}	G_{group}
t_1	J.A.	EU	is	t'_1	J.A.	EU	is	NULL
t_2	O.G.J.	EU	db	t'_2	O.G.J.	EU	db	NULL
t_3	L.C.	Asia	db	t'_3	L.C.	Asia	db	NULL

Such an approach has obvious problems [19]. Assume query Q_2 asks for the name and group of employees from the European Union: $Q_2 = \pi[N, G]\sigma[R = EU]$. Tuples t'_1 and t'_2 pass the selection condition, and the answer to the query is $\{(J.A., NULL), (O.G.J., NULL)\}$. This result does not reveal an essential piece of information, namely that for Jesper Arent (J.A.) no G_{group} attribute exists because he is a member of the information systems group, whereas there exists a group attribute for Ole Guttorm Jensen (O.G.J.) but we do not know its value. (This might be OK for displaying results. It easily leads to major inconsistencies if such results are processed further.) To distinct the two tuples it is possible to refine the semantics of the null value and distinguish between a value that indicates that the attribute is inapplicable and a value that indicates that the value is unknown [18]. Essentially, such an approach uses attribute values to encode schema information. It is problematic to blur the difference between schema and instance information, and we require that no attribute values encode schema information.

2.1 Conditional Schema Changes and Tuple Versioning

In order to accommodate the three requirements discussed above we introduce *conditional schema changes* and *tuple versioning*. They are explored in detail in Sections 3, 4 and 5. This section summarizes the main ideas and illustrates them on an example, which we use throughout the paper.

Conditional schema changes allow schema changes to be applied selectively (requirement R1). A conditional schema change consists of an actual schema change and a condition for applying it. For example, sub-dividing the database unit into groups is a conditional schema change that adds a G_{group} attribute to E_{employee} on condition $U = db$. Conditional schema changes can be decomposed into *conditional schema change primitives*. The conditional schema change that splits the N_{name} of employees from the European Union into F_{first} and L_{last} names can be decomposed into six primitives with the condition $C = EU$: adding attributes

F and L , deleting attribute N , and adding attribute mappings from $\{N\}$ to F , $\{N\}$ to L , and $\{F, L\}$ to N .

Requirements R2 and R3 are accommodated by *tuple versioning*, where each tuple is associated with a *conceptual schema* and a *recorded schema*, respectively. The recorded schema of a tuple is the schema that was used when the tuple was added to the database. The conceptual schema of a tuple denotes the schema a tuple is supposed to have. The recorded schema of a tuple is never affected by schema changes. In contrast, the conceptual schema of a tuple is changed whenever a conditional schema change applies to the tuple. Tuple versioning manages schema changes at a finer granularity than schema versioning. This does not force the database system to globally resolve mismatches by updating tuples in response to schema modifications. Instead, queries can be answered selectively and accurately. Legacy updates remain valid, and the domains of attributes need not be extended with special values for non-existing attributes.

Figure 1 summarizes our solution and shows the evolving E_{employee} schema after applying two schema modifications: sub-dividing the database unit into groups, and splitting the N_{ame} of employees from the European Union into F_{irst} and L_{ast} names.

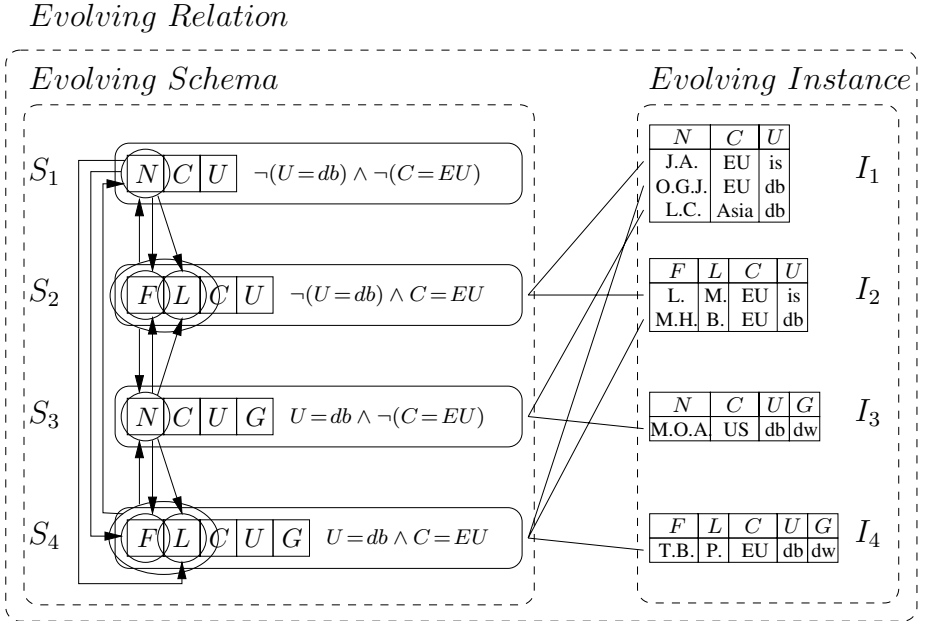


Fig. 1. The Evolving E_{employee} Relation

The *evolving E_{employee} relation* consists of the evolving E_{employee} schema and the evolving E_{employee} instance. The *evolving E_{employee} schema* is defined in terms of a set of *schema segments*, S_1, \dots, S_4 , and corresponding *attribute mappings*.



Each schema segment consists of a schema and a qualifier. As usual, a *schema* is defined in terms of a set of attributes. For example, the schema of segment S_2 is $\{F, L, C, U\}$. The *qualifier* determines which tuples *conceptually* belong to a particular segment. This relationship is indicated by the lines between the tuples and the segments in Figure 1. For example, tuples with $U = db$ and $C \neq EU$ belong to S_3 . In our example this is the tuple in instance I_3 and the third tuple in instance I_1 . The arrows between attributes in different segments are *attribute mappings*. For example, the attribute mapping from N to F indicates that N can be mapped to F . Therefore, if a query asks for the F_{first} name of employees, we can directly answer that query for S_2 and S_4 , and indirectly for S_1 and S_3 , using the attribute mapping from N to F .

We use the conceptual schema of a tuple to determine *whether* a given query can be applied to the tuple, and the recorded schema to determine *how* to apply the query. Consider the query $Q_2 = \pi[N, G]\sigma[C = EU]$, which retrieves name and group of all employees from the European Union. To answer Q_2 , we try to apply the query to each segment in turn. The query cannot be applied to S_1 and S_2 , because neither segment has a G_{group} attribute and there are no attribute mappings that allow to derive the G_{group} attribute. Q_2 can be applied to S_3 . Since only non-European employees qualify for S_3 and the query selects European employees, S_3 does not contribute to the result. This leaves S_4 . Although, S_4 does not contain a N_{name} attribute, Q_2 can be applied to it, because there is an attribute mapping that maps F_{first} and L_{last} to N_{name} . As shown in Figure 1, the tuples associated with S_4 are recorded in different relations. The tuples in instances I_1 and I_2 are not recorded with a G_{group} attribute. Therefore, only N_{name} is projected. The tuple in instance I_4 lacks a N_{name} attribute, but using the aforementioned attribute mapping F_{first} and L_{last} are mapped to N_{name} . Table 2 shows the answer to the query.

Table 2. The Answer to Query Q_2

$N/O.G.J.$	
$N/M.H.B.$	
$N/T.B.P.$	G/dw

3 Evolving Relations

An *evolving schema* $E = (\mathcal{S}, \mathcal{M})$ is defined in terms of a set of *schema segments*, $\mathcal{S} = \{S_1, \dots, S_n\}$, and a set of *attribute mappings*, $\mathcal{M} = \{M_1, \dots, M_n\}$. A segment, $S = (\mathcal{A}, P)$, consists of a schema \mathcal{A} and a qualifier P . As usual, a *schema* is defined as a set of attributes: $\mathcal{A} = \{A_1, \dots, A_n\}$. We write \mathcal{A}_S to denote the schema of segment S . An *attribute constraint* is a predicate of the form $A\theta c$ where A is an attribute, $\theta \in \{<, \leq, =, \neq, \geq, >\}$ is a comparison predicate, and c is a constant. If C is an attribute constraint then $\neg(C)$ is also an attribute constraint. A *qualifier* is either a conjunction of attribute constraints, TRUE, or FALSE.

Let \mathcal{A} be a set of attributes, A be an attribute, and f be a total function such that $f(\mathcal{A}) = A$ and $A \notin \mathcal{A}$. An *attribute mapping*, $S = (\mathcal{A}, A, f)$, establishes a mapping from \mathcal{A} to A .

A tuple t is a set of attributes where each attribute is a name/value pair: $\{A_1/v_1, \dots, A_n/v_n\}$. The value must be an element of the domain of the attribute, i.e., if $\text{dom}(A_i)$ denotes the domain of attribute A_i then $\forall A, v (A/v \in t \Rightarrow v \in \text{dom}(A))$. An *instance* I is a set of tuples with the exact same attribute set, i.e., let \mathcal{A}_I denote the schema of instance I then $\forall t, A' (t \in I \wedge A' \in \mathcal{A}_I \Leftrightarrow \exists A, v (A/v \in t \wedge A = A'))$. An *evolving instance* $\mathcal{I} = \{I_1, \dots, I_n\}$ is a set of instances. We say that a tuple t is an element of an evolving instance \mathcal{I} , $t \in \mathcal{I}$, if there exists an instance $I \in \mathcal{I}$ with $t \in I$.

Definition 1. (evolving relation) Let $E = (\mathcal{S}, \mathcal{M})$ be an evolving schema, $S = (\mathcal{A}, P)$ be a version, \mathcal{I} be an evolving instance. $R = (E, \mathcal{I})$ is an evolving relation iff $\forall S \in \mathcal{S} \exists I \in \mathcal{I} (\mathcal{A}_S = \mathcal{A}_I)$.

Thus, for each segment in an evolving schema there must exist an instance with the exact same schema. This ensures that all tuples can be faithfully recorded, i.e., exactly as specified by the application, and that the structure of recorded tuples never has to be updated.

In an evolving relation, each tuple t has two schemas. The *recorded schema* is the schema of the instance in which t is recorded. The *conceptual schema* is the schema of the version with the qualifier that t satisfies.

In order to avoid undefined qualifiers because of missing attributes, e.g., attributes that were deleted by a schema change, all attributes are implicitly existentially quantified. Thus, $A\theta c$ is an abbreviation for $\exists A \in \mathcal{A} (A\theta c)$, and $\neg(A\theta c)$ is an abbreviation for $\neg \exists A \in \mathcal{A} (A\theta c)$. Note that from this it follows directly that the qualifiers $\neg(A = c)$ and $A \neq c$ are not equivalent. □

Example 1. We use the evolving E_{employee} relation from Figure 1 to illustrate the definitions.

- $E_{\text{employee}} = (\mathcal{S}, \mathcal{M})$ is an evolving schema with four segments, $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$, and three attribute mappings, $\mathcal{M} = \{(\{F, L\}, N, \text{conc}), (\{N\}, F, \text{splitF}), (\{N\}, L, \text{splitL})\}$.
- The attribute mapping $\{(\{F, L\}, N, \text{conc})$ maps F and L to N . Thus, if a query asks for the N_{ame} of employees, we can directly answer that query for segments with a N_{ame} attribute and indirectly for segments with a F_{irst} and L_{ast} attribute.
- Segment $S_3 = (\{N, C, U, G\}, U = db \wedge \neg(C = EU))$ states that the schema for employees in the database unit and not coming from the EU is $\{N, C, U, G\}$.
- The tuple $t = (J.A., EU, is)$ is an element of instance I_1 . Thus, its recorded schema is $\{N, C, U\}$. Because Jesper Arent (J.A.) comes from the EU and because he does not work in the database unit the tuple satisfies the qualifier $\neg(U = db) \wedge C = EU$. Therefore, the tuple qualifies for segment S_2 and its conceptual schema is $\{F, L, C, U\}$.

4 Conditional Schema Change

This section defines conditional schema changes in terms of three primitives, and illustrates how a wide range of schema change operations proposed in the literature can be expressed using conditional schema changes.

A *conditional schema change* is an operation that changes the set of attributes and attribute mappings in an evolving schema. It consists of a list of *conditional schema change primitives* and a condition C . A condition is an attribute constraint, TRUE, or FALSE. We consider three conditional schema change primitives: adding an attribute, α_A , deleting an attribute, β_A , and adding an attribute mapping, $\gamma_{f(A)=A}$. (No primitive is provided to delete attribute mappings. Such an operation can be added easily, and could be used to undo or correct previously added attribute mappings.) These three primitives are sufficient to define changes ranging from simple attribute renamings to advanced splitting and merging of attributes. For example, the change that splits N_{ame} into F_{irst} and L_{ast} name consists of two attribute additions: α_F and α_L , an attribute deletion: β_N , and three additions of attribute mappings: $\gamma_{\text{split}F(\{N\})=F}$, $\gamma_{\text{split}L(\{N\})=L}$, and $\gamma_{\text{conc}(\{F,L\})=N}$.

A consequence of requirement R2 (transparent schema changes) is that schema changes are applied to evolving schemas rather than individual segments. For example, consider the evolving E_{mployee} schema with the segments $S_1 = (\{N, C, U\}, \neg(C = EU))$ and $S_2 = (\{F, L, C, U\}, C = EU)$. When adding a G_{roup} attribute on condition $U = db$, the schema change is applied to both segments. This is quite natural because adding a G_{roup} attribute is independent of and orthogonal to the name of the employees.

The semantics of the three conditional schema change primitives is defined next.

Adding an attribute. An attribute A is added to the schemas of all segments that do not already include the attribute. For each such segment two new segments are generated: a segment with a schema that does not include the new attribute (last line), and a segment with a schema that includes the new attribute (2^{nd} line). Segments with a schema that already includes A are not changed (1^{st} line).

$$\begin{aligned} \alpha_A((\mathcal{S}, \mathcal{M}), C) &= (\mathcal{S}', \mathcal{M}) \text{ iff} \\ \mathcal{S}' &= \{(\mathcal{A}, P) \mid (\mathcal{A}, P) \in \mathcal{S} \wedge A \in \mathcal{A}\} \cup \\ &\quad \{(\mathcal{A} \cup \{A\}, P \wedge C) \mid (\mathcal{A}, P) \in \mathcal{S} \wedge A \notin \mathcal{A}\} \cup \\ &\quad \{(\mathcal{A}, P \wedge \neg C) \mid (\mathcal{A}, P) \in \mathcal{S} \wedge A \notin \mathcal{A}\} \end{aligned} \tag{1}$$

Deleting an attribute. An attribute A is deleted from the schemas of all segments that include the attribute. For each such segment two new segments are generated: a segment with a schema that still includes the attribute (last line), and a segment with a schema that does not include the attribute (2^{nd} line). Segments with a schema that does not include A are not changed (1^{st} line).

$$\begin{aligned}
\beta_A((\mathcal{S}, \mathcal{M}), C) = (\mathcal{S}', \mathcal{M}) \text{ iff} \\
\mathcal{S}' = \{(\mathcal{A}, P) \mid (\mathcal{A}, P) \in \mathcal{S} \wedge A \notin \mathcal{A}\} \cup \\
\{(\mathcal{A} \setminus \{A\}, P \wedge C) \mid (\mathcal{A}, P) \in \mathcal{S} \wedge A \in \mathcal{A}\} \cup \\
\{(\mathcal{A}, P \wedge \neg C) \mid (\mathcal{A}, P) \in \mathcal{S} \wedge A \in \mathcal{A}\}
\end{aligned} \tag{2}$$

We say that a schema change does not apply to a segment if it attempts to add an existing attribute or delete a non-existing one. (According to the above definitions, a schema change that does not apply to a segment leaves the segment unchanged.)

Adding an attribute mapping. An attribute mapping is added unconditionally to the set of attribute mappings:

$$\gamma_{f(\mathcal{A})=A}((\mathcal{S}, \mathcal{M}), C) = (\mathcal{S}, \mathcal{M}') \text{ iff } \mathcal{M}' = \mathcal{M} \cup \{(\mathcal{A}, A, f)\} \tag{3}$$

Example 2. Let segment S_1 have schema $\{N, C, U\}$ and segment S_2 have schema $\{F, L, C, U\}$. Usually, F_{first} and L_{ast} can be concatenated to N_{ame} : $\text{conc}(F, L) = N$. This is expressed in terms of an attribute mapping $M_1 = (\{F, L\}, N, \text{conc})$. Another attribute mapping, $M_2 = (\{N\}, F, \text{splitF})$, might state that F_{first} name can be extracted from N_{ame} : $\text{splitF}(N) = F$.

In contrast to an attribute addition or deletion, adding an attribute mapping is usually not an operation that is directly available to applications. Instead, attribute mappings are specified as parts of schema changes. For example, when changing N_{ame} to F_{first} and L_{ast} , some attribute mappings will be specified along with the change to establish a relationship between the segments. Attribute mappings greatly increase the potential to homogeneously query evolving relations. Therefore, they should be designed carefully.

Definition 2. (conditional schema change) Let E , E' , and E'' be evolving schemas, C be a condition, and $\mathcal{G} = [g_1, \dots, g_n]$ be a list of schema change primitives. Then $\Gamma(E, \mathcal{G}, C) = E''$ is a conditional schema change iff

1. $E' = (\mathcal{S}', \mathcal{M}') = g_n(\dots g_1(E, C) \dots, C)$
2. $E'' = (\mathcal{S}'', \mathcal{M}'')$ with $\mathcal{S}'' = \{(\mathcal{A}, P) \mid (\mathcal{A}, P) \in \mathcal{S}' \wedge P \neq \text{FALSE}\}$ and $\mathcal{M}'' = \mathcal{M}'$

Example 3. Consider the conditional schema change that splits the name of employees from the European Union into first and last names. This schema change can be decomposed into six conditional schema change primitives: adding F_{first} and L_{ast} attributes: α_F and α_L , deleting the N_{ame} attribute: β_N , and adding three attribute mappings: $\gamma_{\text{splitF}(N)=F}$, $\gamma_{\text{splitL}(N)=L}$, and $\gamma_{\text{conc}(F,L)=N}$. Thus, the schema change is defined as follows:

$$\Gamma(E, [\alpha_F, \alpha_L, \beta_N, \gamma_{\text{splitF}(N)=F}, \gamma_{\text{splitL}(N)=L}, \gamma_{\text{conc}(F,L)=N}], C = EU)$$

In the second step of Definition 2 we eliminate segments with false qualifiers to get the intuitively correct result. Assume a conditional schema change that adds attributes A_2 and A_3 on condition C . We expect that each segment to which the change is applied results in two segments: a segment with the same schema and a qualifier extended with the negated condition, and a segment with a schema that includes the new attributes and a qualifier extended with the condition. The sequential application of the schema change primitives in the first step of Definition 2 does not directly provide this:

$$\begin{aligned}
 E &= (\{(\{A_1\}, \text{TRUE})\}, \mathcal{M}) \\
 \alpha_{A_2}(E, C) &= (\{(\{A_1\}, \neg C), (\{A_1, A_2\}, C)\}, \mathcal{M}) \\
 \alpha_{A_3}(\alpha_{A_2}(E, C), C) &= (\{(\{A_1\}, \neg C), (\{A_1, A_2\}, \underline{C \wedge \neg C}), \\
 &\quad (\{A_1, A_3\}, \underline{\neg C \wedge C}), (\{A_1, A_2, A_3\}, C)\}, \mathcal{M})
 \end{aligned}$$

Clearly, the underlined qualifiers are false. Removing the segments with false qualifiers (cf. second step in Definition 2) yields the desired result.

To illustrate the generality of conditional schema changes we show how to use them to express a wide range of schema change operations propose in the literature [17]. We consider operations that relate to the evolution of attributes in a single relation. Not included are changes related to keys. For each schema change operation we specify the equivalent sequence of conditional schema change primitives (CSCP) and give an example. Note that an attribute, $A = (L, \text{dom}(A))$, consists of a label L and a domain $\text{dom}(A)$. To be consistent with current practice and keep the syntax simple we do not usually discuss label and domain explicitly. Here, this is necessary because some schema changes specifically modify the attribute domain.

Add an attribute A :

- Equivalent sequence of CSCP: α_A
- Example: Add a G_{group} attribute to the E_{employee} relation
- $\alpha_{G_{\text{group}}}(E_{\text{employee}}, \text{TRUE})$

Deactivate an attribute A : In schema versioning attributes are deactivated rather than deleted to facilitate the undoing of schema change operations. In our case, schema changes, including attribute deletions, are restricted to the conceptual schema, so attribute deletions can be carried out safely.

- Equivalent sequence of CSCP: β_A
- Example: Drop the sex attribute from the employee relation
- $\beta_{\text{sex}}(E_{\text{employee}}, \text{TRUE})$

Reactivate an attribute A : Attributes that have been deleted (deactivated, cf. above) can simply be added again at some later point in time.

- Equivalent sequence of CSCP: α_A
- Example: (Re-)Add the sex attribute to the employee relation
- $\alpha_{\text{sex}}(E_{\text{employee}}, \text{TRUE})$

Expand the attribute domain of A to $\text{dom}(A')$: To expand the domain of attribute A we delete A and add an attribute A' with the same label and an expanded domain. Expanding the domain guarantees that the domain of A is contained in the domain of A' . Thus, an attribute mapping is added to map A to A' (id denotes the identity function).

- Equivalent sequence of CSCP: $\alpha_{A'}, \beta_A, \gamma_{id(A)=A'}$
- Example: Add the moving constant NOW as a timestamp to the *Date* domain.
- $\alpha_{D'}(R, \text{TRUE}), \beta_D(R, \text{TRUE}), \gamma_{id(D)=D'}(R, \text{TRUE})$
if $D = ('Date', \text{dom}(D))$ and $D' = ('Date', \text{dom}(D) \cup \{\text{NOW}\})$ are attributes, and D is an attribute of R

Restrict the attribute domain of A to $\text{dom}(A')$: Restricting the attribute domain is symmetric to expanding the attribute domain. The only difference is the direction of the attribute mapping.

- Equivalent sequence of CSCP: $\alpha_{A'}, \beta_A, \gamma_{id(A')=A}$
- Example: Drop 00 as a possible grade
- $\alpha_{G'}(R, \text{TRUE}), \beta_G(R, \text{TRUE}), \gamma_{id(G')=G}(R, \text{TRUE})$
if $G = ('Grade', \text{dom}(G))$ and $G' = ('Grade', \text{dom}(G) \setminus \{00\})$ are attributes and G is an attribute of R .

Change the attribute domain of A to $\text{dom}(A')$: In the general case, changing the domain of an attribute does not result in attribute mappings. However, in many cases a mapping may exist. In our example, a mapping exists in both directions. The functions $f_{DKK2Euro}$ and $f_{Euro2DKK}$ use the corresponding exchange rates to convert the currencies.

- Equivalent sequence of CSCP: $\alpha_{A'}, \beta_A$
- Example: Change the currency from DKK to Euro
- $\alpha_{S'}(R, \text{TRUE}), \beta_S(R, \text{TRUE}), \gamma_{f_{DKK2Euro}(S)=S'}(R, \text{TRUE})$, and $\gamma_{f_{Euro2DKK}(S')=S}(R, \text{TRUE})$
if $S = ('Sales', \text{dom}(S))$ and $S' = ('Sales', \text{dom}(S'))$ are attributes, and S is an attribute of R .

Rename an attribute A to A' : An attribute renaming affects attribute names (labels) but does not change attribute domains or values. Because only the label is changed attribute mappings in both directions exist.

- Equivalent sequence of CSCP: $\alpha_{A'}, \beta_A, \gamma_{id(A)=A'}, \gamma_{id(A')=A}$
- Example: Rename units to departments in the E_{employee} relation
- $\alpha_D(E, \text{TRUE}), \beta_U(E, \text{TRUE}), \gamma_{id(U)=D}(E, \text{TRUE})$, and $\gamma_{id(D)=U}(E, \text{TRUE})$
if $U = ('Unit', \text{dom}(U))$ and $D = ('Department', \text{dom}(U))$ are attributes, and U is an attribute of E

Conditional schema changes never update the recorded schema of a tuple. However, the conceptual schema of tuples may change. Consider the instance I_1 in Figure 1, and assume that the conceptual schema of all tuples in I_1 is S_1 . The conditional schema change $\alpha_G(E, U = db)$ changes the conceptual schema of tuples satisfying the condition to the schema of segment S_2 . Those tuples that do not satisfy the condition, retain their conceptual schema.

Lemma 1. *Let $R = (E, \mathcal{I})$ be an evolving relation, $t \in \mathcal{I}$ be a tuple with conceptual schema \mathcal{A}_S , where $S \in E$, and let the conditional schema change $\Gamma(E, \mathcal{G}, C) = E'$. The conceptual schema of t is changed iff $S \notin E'$ and t satisfies C .*

Proof. (Sketch) To prove the lemma we consider each schema change primitive in turn. Let S be a segment and \mathcal{A}_S be the conceptual schema of t . By definition, mapping additions do not change the set of segments, so they cannot change the conceptual schema of t . Thus the conceptual schema of t is unchanged. For attribute additions and deletions there are three cases. 1) Γ does not apply to S . In this case, S is propagated to the result directly, $S \in E'$, and the conceptual schema of t is unchanged, if $S \in E'$. 2) \mathcal{A}_S is changed and C is included in the qualifier. Thus, $S \notin E'$ and the conceptual schema of t is changed if t satisfies C . 3) \mathcal{A}_S is unchanged and $\neg(C)$ is included in the qualifier. Thus, $S \notin E'$ and the conceptual schema of t is unchanged, if t does not satisfy C . Therefore, the conceptual schema of t is *only* changed, if $S \notin E'$ and t satisfies C .

5 Querying Evolving Relations

This section presents an algorithm that computes queries over evolving relations. We prove that the set of versions considered by the algorithm is maximal, and that a conditional schema change does not reduce this set.

Queries are asked with respect to a schema. It is the task of the DBMS to determine whether the query actually applies to the schema. E.g., if a query refers to attributes not in the schema, then the query does not apply to that schema, and no answer can be computed. *Logical query answering* determines whether a query applies to a given evolving relation. Specifically, the conceptual schemas to which the query can be applied are identified. *Physical query answering* determines how to apply the query to an evolving relation. Specifically, mismatches between conceptual and recorded schemas are resolved, and the uniformity of the query answer is maximized using attribute mappings.

5.1 Logical Query Answering

With multiple schema segments it is obvious that a query might apply to some segments but not to other ones. Because queries are issued against evolving schemas rather than individual segments we have to define the semantics of a query issued against an evolving schema.

We consider queries of the form $Q = \pi[A_1, \dots, A_n]\sigma[C]$, and write \mathcal{A}_Q to denote the set of attributes used in Q . A query can be *applied* to a schema segment if all attributes in the query also appear in the segment, or if an attribute $A \in \mathcal{A}_Q$ that does not appear in the schema of a segment can be derived from the attributes of the segment and the attribute mappings.

Definition 3. Let A be an attribute, \mathcal{A} be a set of attributes, and \mathcal{M} be a set of attribute mappings. A mapping from \mathcal{A} to A , $\text{map}(\mathcal{A}, A, \mathcal{M})$, is defined recursively as follows:

- $\text{map}(\mathcal{A}, A, \mathcal{M}) = \omega$, iff $A \notin \mathcal{A}$ and $\neg \exists \mathcal{A}_i, f_i((\mathcal{A}_i, A, f_i) \in \mathcal{M})$
- $\text{map}(\mathcal{A} \cup \{A\}, A, \mathcal{M}) = A$
- $\text{map}(\mathcal{A}, A, (\{\mathcal{A}_1, \dots, \mathcal{A}_n\}, A, f) \cup \mathcal{M}) = f(\text{map}(\mathcal{A}, \mathcal{A}_1, \mathcal{M}), \dots, \text{map}(\mathcal{A}, \mathcal{A}_n, \mathcal{M}))$

Example 4. Let $\mathcal{M} = \{(\{N\}, F, \text{split}F), (\{F, L\}, N, \text{conc}), (\{F\}, X, \text{gender})\}$ be a set of attribute mappings. (We assume a function *gender* that, given a first name, determines the sex, X , of a person.) This leads to the following mappings:

1. $\text{map}(\{N\}, L, \mathcal{M}) = \omega$
2. $\text{map}(\{F, L\}, N, \mathcal{M}) = \text{conc}(F, L)$
3. $\text{map}(\{N\}, X, \mathcal{M}) = \text{gender}(\text{split}F(N))$

The MAP algorithm illustrated in Figure 2 implements the mapping. Given a set of source attributes, a target attribute, and a set of attribute mappings, MAP returns a term that maps the source attributes to the target attribute. If no such term exists, the algorithm returns ω .

```

MAP( $\mathcal{A}, A, \mathcal{M}$ ):
  Input:
    set of attributes  $\mathcal{A}$ 
    attribute  $A$ 
    set of attribute mappings  $\mathcal{M}$ 
  Output:
    term  $F(A)$  or  $\omega$ 
  Method:
    if  $A \in \mathcal{A}$  return  $A$ 
    for each  $M_i \in \mathcal{M}$  where  $M_i = (\mathcal{A}_i, A, f_i)$  do
      let  $\text{ExistsMapping} := \text{TRUE}$ 
      for each  $A_j \in \mathcal{A}_i$  do
        let  $f_j := \text{MAP}(\mathcal{A}, A_j, \mathcal{M} \setminus S_i)$ 
        let  $\text{ExistsMapping} := \text{ExistsMapping} \wedge f_j \neq \omega$ 
      rof
      if  $\text{ExistsMapping}$  return  $f(f_1, \dots, f_n)$ 
    rof
  return  $\omega$ 

```

Fig. 2. The MAP Algorithm

Let $E = (\mathcal{S}, \mathcal{M})$ be an evolving schema. A query Q *applies to* a segment $S \in \mathcal{S}$, $Q \succ V$, iff $\forall A \in \mathcal{A}_Q : \text{MAP}(\mathcal{A}_S, A, \mathcal{M}) \neq \omega$. Intuitively, this means that the attributes in segment S can be mapped to each attribute in query Q .

The validity function, $\text{val}(Q, E)$, denotes the set of segments in E that Q can be applied to: $\text{val}(Q, E) = \{S \mid E = (\mathcal{S}, \mathcal{M}) \wedge S \in \mathcal{S} \wedge Q \succ S\}$. A query is *valid* iff it applies to at least one segment of an evolving schema, i.e., $\text{val}(Q, E) \neq \emptyset$.

Example 5. Assume the segments S_1 and S_2 , and the attribute mappings from Figure 1: $\mathcal{M} = \{(\{N\}, F, \text{split}F), (\{N\}, L, \text{split}L), (\{F, L\}, N, \text{conc})\}$. Trivially, $\pi[N](E) \succ S_1$ because S_1 contains the N attribute. $\pi[N](E) \succ S_2$ is true because of the attribute mapping between $\{F, L\}$ and N : $\text{MAP}(\{F, L, C, U\}, N, \mathcal{M}) = \text{conc}(F, L)$.

Lemma 2. *Let E be an evolving schema and let Q be a query over E . $val(Q, E)$ is the maximal set of segments to which Q can be applied.*

Proof. Assume a segment S that is not part of the validity set, $S \notin val(Q, E)$, but to which Q can be applied. We show by contradiction that this is impossible. If Q can be applied to S the attributes in S can be used to derive each attribute used in Q . By definition, this means that $Q \succ S$. Clearly, this is a contradiction because $Q \succ S \Rightarrow S \in val(Q, E)$.

The validity function is monotonic in the sense that conditional schema changes never reduce the set of versions that subsume a query Q . The reason is that conditional schema changes preserve the old schema.

Lemma 3. *Let E and E' be evolving schemas, let Γ_C be a conditional schema change such that $\Gamma_C(E) = E'$, and let Q be a query over E . A conditional schema change does not restrict the set of version to which a query can be applied: $val(Q, E) \subseteq val(Q, E')$.*

Proof. We investigate each of the three schema changes in turn. Adding an attribute to a segment that already includes the attribute simply propagates the segment to E without changing the attribute mappings. Thus, $S \in val(Q, E')$ also holds. If the segment does not yet include the attribute then E' will contain a segment that has the exact same schema as the original segment. Thus, $S \in val(Q, E')$ holds again. The equivalent holds true for attribute deletions. Finally, adding an attribute mapping does not eliminate existing attribute mappings and again $S \in val(Q, E')$ follows.

5.2 Physical Query Answering

A valid query is applied to each segment in the validity set for that query. This means that only tuples with conceptual schemas corresponding to those segments contribute to the query answer. Since the recorded schema may differ from the conceptual schema, the query must be transformed to the recorded schema to answer the query. This requires that the mismatches between the conceptual and the recorded schema are resolved. In particular, attributes present in the conceptual schema but missing in the recorded schema must be dealt with: If a missing attribute is used in a projection it is omitted, and a selection predicate $A\theta c$, with the missing attribute A , is replaced by FALSE (cf. Example 6).

We define a query transformation function $TRANSFORM(Q, E, I)$ that rewrites a query Q over E to the schema of an instance I using the mapping MAP. Figure 3 presents an algorithm for the query transformation function.

Example 6. Assume the query $Q = \pi[N, G]\sigma[C = EU]$ issued against the evolving E_{employee} schema shown in Figure 1. TRANSFORM produces the following rewritten queries:

$Q_{I_1} = \pi[N]\sigma[C = EU]$ During the transformation the G_{group} attribute had to be eliminated from the projection because the tuples in I_1 were recorded without this attribute.

```

TRANSFORM( $Q, E, \mathcal{A}_I$ ):
  Input:
    query  $Q$ 
    evolving schema  $E = (\mathcal{S}, \mathcal{M})$ 
    recorded schema  $\mathcal{A}_I$ 
  Output:
    rewritten query  $Q_I$  that fits the schema of instance  $I$ 
  Method:
    let  $Q_I := Q$ 
    for each  $A_i$  used in  $Q_I$  do
      Let  $f_i := \text{MAP}(\mathcal{A}_I, A_i, \mathcal{M})$ 
      if  $f_i \neq \omega$  replace any occurrence of  $A_i$  in  $Q_I$  with  $f_i$ 
      else /* mismatch between conceptual
           and recorded schema */
        if  $A_i$  appears in a projection  $\pi$ 
          remove any occurrence of  $A_i$  in  $\pi$  from  $Q_I$ 
        if  $A_i$  appears in any predicate
           $P \in \{A_i \theta c, A_i \theta A_j\}$  in a selection  $\sigma$ 
            replace each  $P$  with the constant FALSE
        fi
    rof
    return  $Q_I$ 

```

Fig. 3. The TRANSFORM Algorithm

$Q_{I_2} = \pi[\text{conc}(F, L)]\sigma[C = EU]$ The G_{group} attribute has to be removed for the same reason as above. An attribute mapping has to be used to construct N_{name} from F_{first} and L_{last} names.

$Q_{I_3} = \pi[N, G]\sigma[C = EU]$ All attributes used in query Q are present in the schema of instance I_3 . Thus, $Q_{I_3} = Q$.

$Q_{I_4} = \pi[\text{conc}(F, L), G]\sigma[C = EU]$ An attribute mapping has to be used to construct N_{name} from F_{first} and L_{last} names.

Because $\text{val}(Q, E) = \{S_3, S_4\}$, only tuples with a conceptual schema equal to S_3 or S_4 will be considered for the computation. The combination of the partial results from evaluating the Q_{I_i} 's yields the answer shown in Table 2.

Note that the query transformations are independent of how instances are stored physically. A solution in terms of multiple relations is as well possible as one in terms of a single relation. For example, to apply Q_{I_1} to a storage model consisting of a single relation, e.g. a completed schema, a (brute-force) approach would be to select the tuples with a conceptual schema corresponding to either S_3 or S_4 and a recorded schema corresponding to I_1 , and to apply Q_{I_1} to these tuples.

6 Related Work

Interest in evolving database systems [15] has predominantly resulted from research in temporal databases and in object-oriented databases.

In OODBs, evolution has been investigated with respect to architectures suitable for CAD or other engineering domains [19], and a series of papers address issues in data modeling [3,14,4], architecture [3,12,1,10], and query language support [11]. Evolution of classes in OODEs allows for multiple classifications of instances through class generalizations or specializations. In our framework each tuple has a unique interpretation (via its conceptual schema). This makes it possible to handle segments transparently, so that users do not have to be aware of individual segments when interacting with the DBMS.

In temporal database, schema evolution has been analyzed in the context of temporal data models [7,2]. In the literature, several proposals have been made for the maintenance of schema versions along one [8,13,16,20,21] or more time dimensions [6]. In our work schema changes are not restricted to the time, but can be conditioned by all attributes.

All related work investigates schema versioning where schema changes are applied to individual schema versions. In our framework, individual segments are not first class objects, and schema changes are applied at a higher granularity to the entire evolving schema. The schema changes are then propagated automatically to individual segments.

In schema versioning, the schema and the tuples are managed at the same level [9,5], i.e., schema and instance changes are synchronized. In our framework, we manage versioning at a finer granularity using tuple versioning where each segment has a recorded and a conceptual schema. As a consequence, we avoid data migration and the problems of null values apparent in schema versioning [19].

At the implementation level it is possible to use different techniques to support tuple versioning and conditional schema changes. A potential candidate is the view mechanism of database systems. Each conceptual schema can be expressed as a view over the recorded schemas. Similarly, views can be used to implement attribute mappings. Such a solution does not allow to query evolving schemas homogeneously. To uniformly query evolving schemas we need solutions beyond pure first order logic. The view mechanism is also not well-suited to resolve mismatches between conceptual and recorded schemas. The view mechanism is insufficient if attributes that are not present in the recorded schemas shall lead to heterogeneous result tuples (cf. Table 2).

7 Conclusions and Future Research

We proposed and formalized a framework for evolving relations defined by evolving schemas and corresponding evolving instances. Our framework allows for transparent and selective schema modifications using *conditional schema changes*, which are formalized in terms of three primitives. We introduced *tuple versioning*, where each tuple has a recorded and a conceptual schema. Doing so we avoid the problems of value-encoded schema information, and we showed that only the conceptual schema needs to be updated in response to schema changes. We presented an algorithm to answer queries over evolving relations, and argued that the maximal set of tuples is considered.

Our work suggests several lines of future research. Different strategies for physically representing evolving relations could be analyzed and measured. In particular, the representation of evolving instances is not restricted. A representation in terms of a single instance is as well possible as a representation in terms of multiple instances. It would also be interesting to develop indexing strategies based on a numbering of recorded and conceptual schemas, to facilitate efficient processing of queries over evolving relations. Finally, we consider specializing our framework to time, to exploit some of the unique properties of time such as the strict ordering of schema changes.

References

1. J. Andany, M. Leonard, and C. Palisser. Management of schema evolution in databases. In *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 161–170. Morgan Kaufmann, 1991.
2. G. Ariav. Temporally oriented data definitions: managing schema evolution in temporally oriented databases. *Data Knowledge Engineering*, 6(6):451–467, 1991.
3. J. Banerjee, W. Kim, H.-J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD International Conference on Management of Data*, pages 311–322. ACM Press, 1987.
4. B. Benatallah. A unified framework for supporting dynamic schema evolution in object databases. In *ER '99, 18th International Conference on Conceptual Modelling, Paris, France, November 15-18, 1999, Proceedings*, pages 16–30. Springer, 1999.
5. C.D. Castro, F. Grandi, and M.R. Scalas. *On schema versioning in temporal databases. In: Recent Advances in Temporal Databases*. Springer, 1995.
6. C.D. Castro, F. Grandi, and R.R. Scalas. Schema versioning for multitemporal relational databases. *Information Systems*, 22(5):249–290, 1997.
7. J. Clifford and A. Croker. The historical relational data model (HRDM) and algebra based on lifespans. In *3rd International Conference of Data Engineering, Los Angeles, California, USA, Proceedings*, pages 528–537. IEEE Computer Society Press, 1987.
8. P. Dadam and J. Teuhola. Managing schema versions in a time-versioned non-first-normal-form relational database. In *Datenbanksysteme in Büro, Technik und Wissenschaft, Darmstadt, West Germany, Proceedings*, pages 161–179. Springer-Verlag, 1987.
9. R.T. Snodgrass et al. TSQL2 language specification. *ACM SIGMOD Record*, 23(1), 1994.
10. F. Ferrandina, T. Meyer, R. Zicari, Guy Ferran, and J. Madec. Schema and database evolution in the O2 object database system. In *21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland, Proceedings*, pages 170–181. Morgan Kaufmann, 1995.
11. F. Grandi and F. Mandreoli. ODBM language extensions for generalised schema versioning support. In *ER '99 Workshops on Evolution and Change in Data Management, Paris, France, November 15-18, 1999, Proceedings*, pages 36–47. Springer, 1999.
12. W. Kim and H.-T. Chou. Versions of schema for object-oriented databases. In *14th International Conference on Very Large Databases, Los Angeles, California, USA, Proceedings*, pages 148–159. Morgan Kaufmann, 1988.

13. L.E. McKenzie and R.T. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.
14. S.R. Monk and I. Sommerville. Schema evolution in OODBs using class versioning. *SIGMOD Record*, 22(3):16–22, 1993.
15. J.F. Roddick. Schema evolution in database systems an annotated bibliography. *ACM SIGMOD Record*, 21(4):35–40, 1992.
16. J.F. Roddick. SQL/SE - a query language extension for databases supporting schema evolution. *ACM SIGMOD Record*, 21(3):10–16, 1992.
17. J.F. Roddick. Implementing schema evolution in relational database systems: An approach based on historical schemata. Technical report, Dept. of Computer Science and Computer Engineering, La Trobe University, 1993.
18. J.F. Roddick. *A model for temporal inductive inference and schema evolution in relational database systems*. PhD thesis, La Trobe University, 1994.
19. J.F. Roddick. A survey of schema versioning issues for database systems. *Information Software Technology*, 37(7):383–393, 1995.
20. J.F. Roddick, N.G. Craske, and T.J. Richards. A taxonomy for schema versioning based on the relational and entity relationship models. In *12th International Conference on Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993, Proceedings*, pages 137–148. Springer-Verlag, 1993.
21. J.F. Roddick and R.T. Snodgrass. *Schema versioning*. In: *The TSQL92 Temporal Query Language*. Noewell-MA: Kluwer Academic Publishers, 1995.
22. D. Sjöberg. Quantifying schema evolution. *Information Software Technology*, 35(1):35–44, 1993.

QFD Matrix for Incremental Construction of a Warehouse via Data Marts

Ron McFadyen and Fung-Yee Chan

University of Winnipeg, Winnipeg, Canada
mcfadyen-r@c-h.uwinnipeg.ca

Abstract. In this paper we consider the construction of a dimensional data warehouse. The warehouse is built beginning with the first data mart and proceeding in an iterative manner constructing one mart at a time. In this way the warehouse is seen to evolve over time. This evolutionary process is necessary due to the complexity of data stores, relationships, transformations, and the processing involved. In this paper we consider the problem of identifying the next data mart to construct and present a tool based on Quality Function Deployment for use in the planning stages.

1 Introduction

Many enterprises have developed substantial information systems. At the core of these systems are operational databases that support the processing of information that is used to run the day-to-day operations of the enterprise. In general, these operational databases are complex, heterogeneous, large, and have evolved over many years (even decades). To address issues related to planning and decision making, many enterprises are just beginning to create data warehouse systems to facilitate the analysis of data by decision-makers.

In Figure 1 we illustrate the basic architecture of a data warehousing system (similar to that given in [12]). We consider the data warehouse to be a collection of data marts where each data mart is oriented to a specific area of the enterprise. Each data mart is designed and constructed as a multidimensional model; the Dimensional Fact model proposed in [4] would be appropriate for designing a data mart. A common representation of a multidimensional model is the Star Schema, popularized by [9].

A star schema is a constrained database design where one table, the fact table, participates in many one-to-many relationships with other tables referred to as dimensions; each relationship involves the fact table and a dimension table. The fact table can also be considered to represent the many-to-many-to-many-to-...-many relationship amongst the dimensions. See Figure 2 for an example of a Sales schema where the measurements (the facts) recorded for each sale are "Dollar sales" and "Units sales". Each pair of measurements is related to one store, one product, one day, and one promotion record. This schema enables the decision-maker to analyze sales to discover trends and other information that

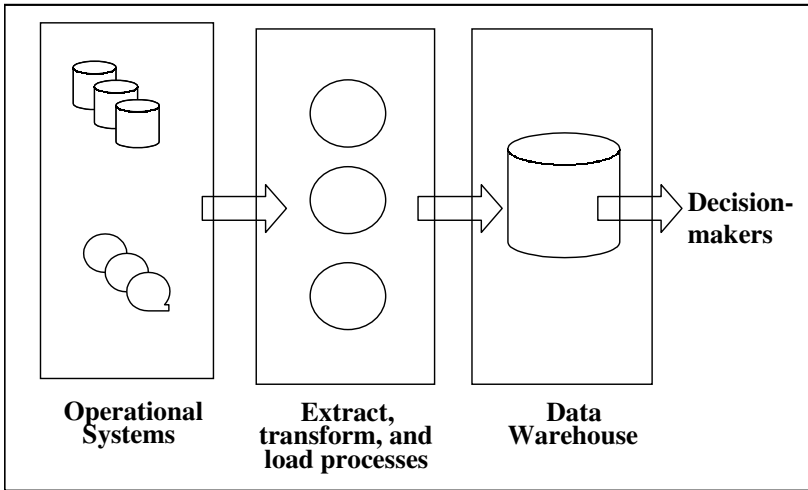


Fig. 1. Data warehousing architecture

may assist in increasing sales. An underlying assumption of our warehouse model is that the dimensional model is an appropriate data model for a decision-maker or business analyst.

To construct the warehouse we are required to construct the data marts. It is generally considered that doing this all at once, in a single project, is not feasible; a guiding principle given in [7] is "data warehouse development is an iterative process". The construction of the warehouse is an iterative or evolutionary process. In [4] a methodology is proposed for developing the data warehouse model; this methodology begins with defining all the facts. In practice, we will not know all the facts that are to be placed in the warehouse. The set of facts required by the enterprise will unfold over time, and the time can be measured in years. In this paper we are concerned with this iterative aspect for evolving the data warehouse. In particular, we are concerned with the process we use to select the next data mart to construct.

2 The Matrix

In [10] the Matrix is given as a planning tool for guiding the construction of the warehouse. The Matrix comprises a vertical list of data marts and a horizontal list of dimensions. The Matrix indicates all the dimensions that each data mart will need. The information in the Matrix is obtained through analysis techniques such as interviews and group sessions. We reproduce a portion of the example Matrix from [10] in Figure 3.

The list of data marts represents a set of data marts that can take several years to construct. The first mart must be chosen, then the second and so on.

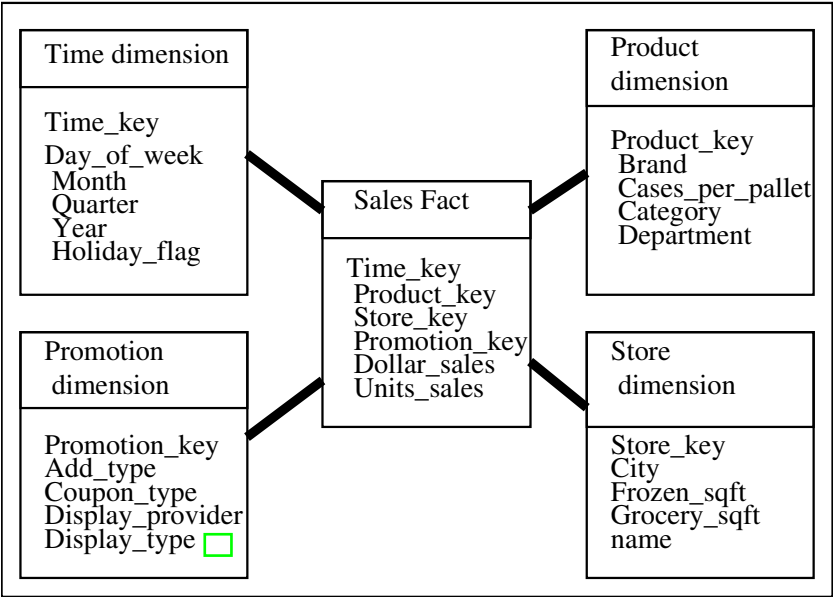


Fig. 2. Star Schema

[10] recommends to begin construction of the warehouse with the data mart that represents the least amount of effort and risk. A first data mart will create a foundation on which others will be built. Dimensions will be created that will be reused by other data marts simplifying the building of those subsequent marts.

We can see a lot of useful information by examining the Matrix, but by incorporating quality function deployment principles we can add value to it, value that helps us demonstrate some planning conclusions. In particular, how do we determine the first data mart to be built, the second, etc.?

The cost of building the warehouse is the cost of building all the data marts. To build a data mart we must build all the necessary dimensions, the fact tables, and the extract, transform and load routines. We cannot use an arbitrary sequence though. If we choose poorly we may have a very difficult initial task that will doom us because it took too long, cost too much, and the warehouse project gets cancelled. We need to choose in such a way that we obtain early successes, gain user acceptance, and gain experience. Choosing a next data mart that is technically feasible and which meets some political acceptance is crucial.

3 Quality Function Deployment

Quality Function Deployment (QFD) has been used since the 1960's and has been adopted by many large corporations [6]. [5] describes a survey of major software vendors that have adapted QFD for the requirements gathering phase of the System Development Lifecycle (SDLC). This adaptation of QFD is termed

<div>Data mart</div>	Time	Customer	Calling Party	Called Party	Supplier
Customer billing	X	X			
Trouble reporting	X	X	X		X
Yellow pages ads	X	X	X		
Billing call detail	X	X	X	X	X

X implies inclusion of dimension in data mart

Fig. 3. The Matrix

Software Quality Function Deployment (SQFD). Many quality improvements are attributed to SQFD, including increased analyst and programmer productivity, fewer design changes, and less maintenance. [5] does not describe specific implementations of SQFD. [2] shows how QFD techniques can be applied to the SDLC. QFD matrices have been adapted to object oriented methodologies in [3]. Data warehouse quality and QFD are discussed in [8], where they are concerned with the quality of schema design and the quality of the data inserted into the warehouse, but not on the quality of the process used to determine which schema to design next.

At the heart of QFD is a matrix-like structure bearing some resemblance to the Matrix presented above. The QFD structure we will discuss is shown in Figure 4 where:

- Area A contains key customer requirements
- Area B contains key product characteristics corresponding to the requirements
- Area C is the relationship between the customer requirements and the product characteristics
- Area D gives more information about the product characteristics such as relative cost and degree of technical difficulty.
- Area E, the roof matrix, shows the relationship between different product characteristics

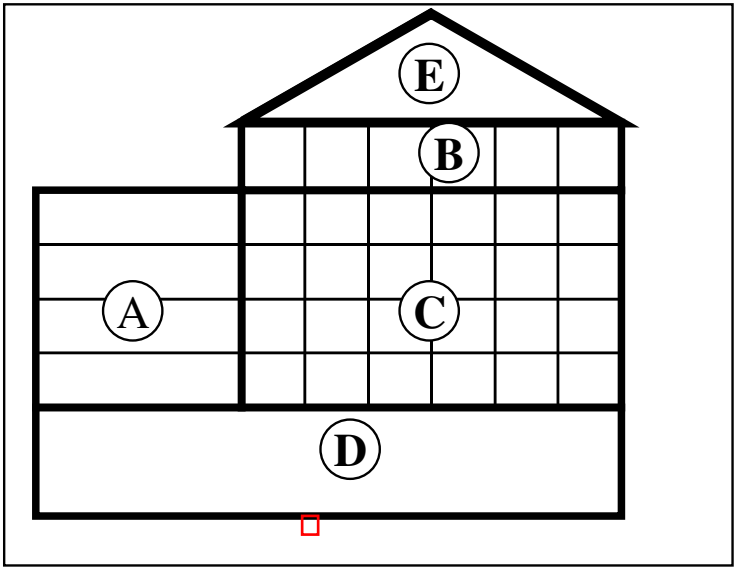


Fig. 4. QFD Structure

In Figure 5 we illustrate a portion of a QFD matrix from [6] that illustrates considerations arising in the design of a pencil. From the QFD matrix we note:

- The customer considers clear black line and point lasts as the most important characteristics; these are most strongly influenced by the lead dust generated and the time between sharpening characteristics.
- The two product characteristics, lead dust generated and the time between sharpening, have negative influences on each other (the X in the roof). To create a product with high values for these characteristics will present challenges to the engineers.
- The two product characteristics, lead dust generated and the time between sharpening, are each technically challenging. To create a product with high values for one of these is more challenging than creating a product with a long length.
- Cost is not necessarily directly associated with technical difficulty; other factors may come into play. Although a high value for lead dust generated is technically difficult to achieve, it has the smallest relative cost shown.

These points are easily observed in the QFD diagram and help to convey reasoning for product strategy going forward. In the next section we discuss how QFD can be applied to the Matrix to facilitate the warehouse planning process.

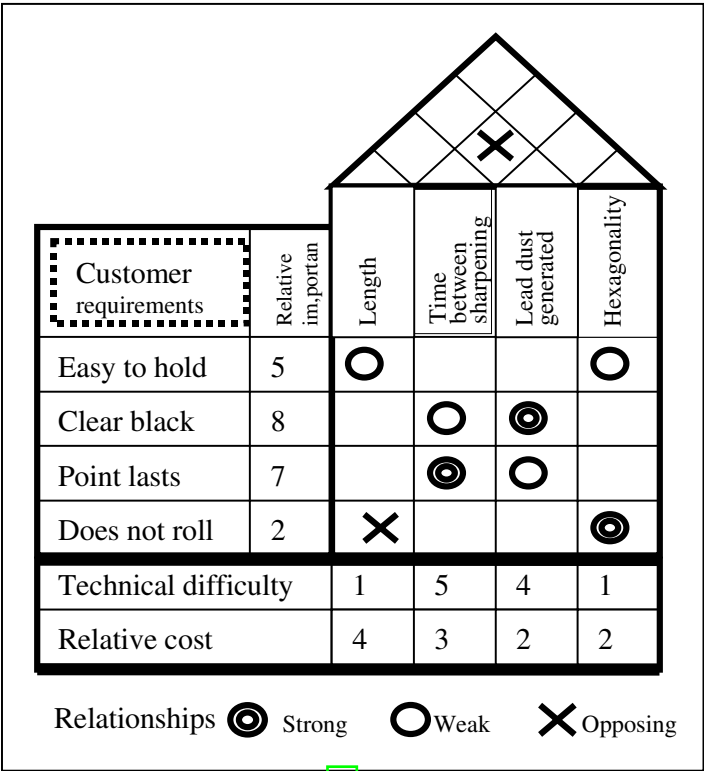


Fig. 5. A QFD Example

4 The DWQFD Matrix

The Matrix in [10] is useful for illustrating the data marts that need to be built to create the data warehouse. A QFD diagram is useful for capturing information that justifies a product strategy. Here, we combine these two structures to help capture information that is useful for planning the deployment of the data marts; we refer to the combined structure as the DWQFD Matrix.

Consider Figure 6 which captures information regarding:

- The data marts to be constructed and their pre-assigned relative importance (perhaps decided to a large degree by company politics). (Area A)
- The dimensions that need to be constructed. (Area B)
- The dimensions required for each data mart. With QFD relationships we can capture the degree of confidence we have for whether or not a dimension is expected to be part of a data mart. If there is strong agreement regarding the usefulness of a dimension to a data mart, or disagreement regarding its need. (Area C)

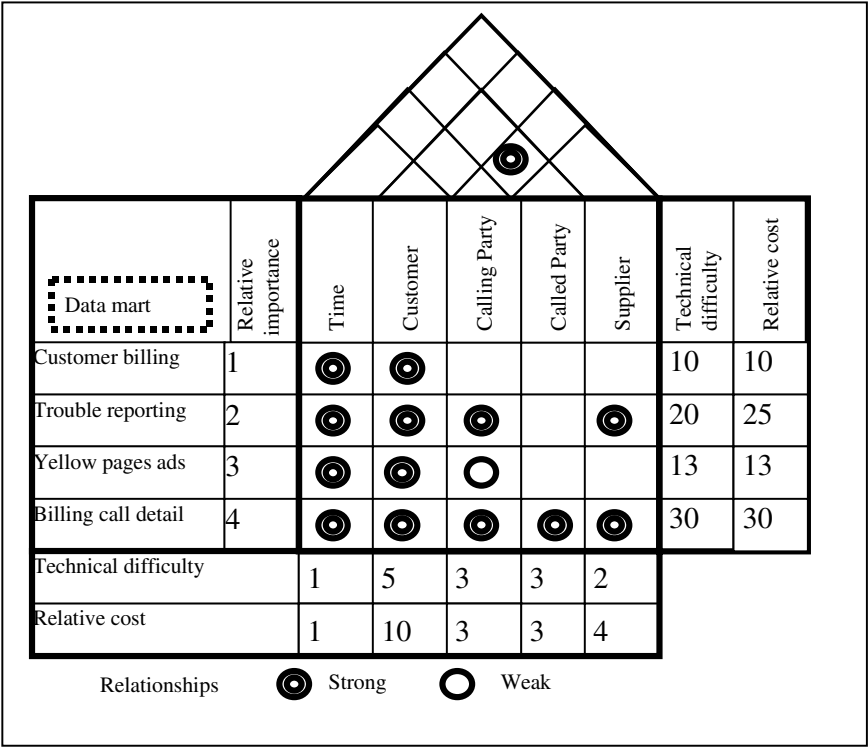


Fig. 6. The DWQFD Matrix

- The roof matrix captures associations that may exist between dimensions. Some dimensions will be views of others; some will be sourced from the same legacy tables. (Area E)
- The technical difficulty of constructing a dimension is represented. A time dimension may be technically the easiest, whereas a customer dimension may be extremely difficult. The customer dimension may be sourced from several tables in the legacy environment that evolved over a period of decades and for which documentation is not to be found, or may not be consistent. (Area D)
- The relative cost of constructing dimensions is represented. Cost will be related to factors such as the technical difficulty, the number of source tables, the complexity of joins (or matching), and the complexity of transformations. (Area D)
- The relative cost and technical difficulty for the construction of each data mart can be estimated and added to the matrix; the relative cost and technical difficulty of each dimension (Area D), and the relationships between dimensions (Area E), contribute to these estimates. This information along

with political factors can be used to justify which data mart should be tackled next. (A new area to the right of Area C)

In the DWQFD Matrix of Figure 6 we have captured more information than would be present in the Matrix alone. The DWQFD Matrix documents the following:

- There is a high correlation between the Calling Party and the Called Party dimensions. It is assumed here that these are sourced from the same legacy tables and would in fact be represented by the same physical dimension. Hence by constructing one, the other is automatically created.
- The user community does not have a strong consensus for whether or not the Yellow Pages Ads data mart will need the Calling Party dimension.
- The Customer dimension is the most difficult to build, and the Time dimension the easiest. The Supplier dimension is perceived to be less difficult to build than the Calling Party dimension.
- Although the Supplier dimension is less challenging technically, it is considered more costly to build than the Calling Party dimension. A reason for this could be because developer time required for the Supplier dimension is substantially more than that for the Calling Party dimension, which could be related to the type of source databases involved.

The information documented in the DWQFD Matrix can be used to justify and support the data mart construction strategy. The Customer Billing data mart is still seen to be the least challenging and the least costly, so it should still be ranked number one and be the first to be constructed. However, from the information shown the second data mart should likely be the Yellow Pages Ads data mart. It represents a lesser technical challenge and its cost is less than the Trouble Reporting data mart (initially ranked second). When conveying this analysis and when making recommendations to management, the warehouse administrator has the appropriate tool for representing information supporting his or her strategy for warehouse construction.

5 Summary

The warehouse construction process is evolutionary due to the huge scope that it represents and due to the large number of resources required to bring it from planning to realization. The process is iterative; each phase begins with the choice of the next data mart. The DWQFD Matrix is a useful tool to guide this process. At the initiation of each phase, the DWQFD Matrix is re-evaluated to determine which data mart is the best choice at that point in time. Once one data mart has been built, the relative cost and technical challenge of many others are likely to change. This is because dimensions will be reused, and because the project team has obtained experience with the data, the transformations, and the technology.

In this paper we have adapted the QFD to the Matrix generating a new structure, the DWQFD Matrix, which is a useful tool for planning the incremental/evolutionary warehouse construction.

References

1. Adamson, Christopher, Venerable, Michael: Data warehouse design solutions. John Wiley & Sons, Inc. ISBN 0-471-25195-X (1998).
2. Inger V. Eriksson, Fred McFadden, Anne M. Tiittanen: Improving software development through quality function deployment. Proceedings of the Fifth International Conference on Information Systems Development (1996).
3. Walter M. Lamia: Integrating QFD with object oriented software design methodologies. QFD Symposium (1995).
4. Matteo Golfarelli, Dario Maio, Stefano Rizzi: Conceptual Design of data warehouses from E/R schemes. Proceedings of the Hawaii International Conference on System Sciences (January 1998).
5. Stephen Haag, M. K. Raja, L. L. Schkade: Quality Function Deployment usage in software development. Communications of the ACM (39) 1 (January, 1996).
6. David Hutton: Quality Function Deployment.
<http://www.dhutton.com/visitors/articles/artqfd.html>.
7. W. H. Inmon, John A. Zachman, Jonathon G. Geiger: Data stores, data warehousing and the Zachman framework. McGraw-Hill. ISBN 0-07-031429-2 (1997).
8. Matthias Jarke, Manfred A. Jeusfeld, Christoph Quix, Timos Sellis, Panos Vassiliadis: Chapter 7, Metadata and data warehouse quality, from Fundamentals of data warehouses by Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, Panos Vassiliadis. Springer-Verlag. ISBN 3-540-65365-1 (2000).
9. Ralph Kimball: The data warehouse toolkit. John Wiley & Sons, Inc. ISBN 0-471-15337-0 (1996).
10. Ralph Kimball: The matrix. The Intelligent Enterprise. December 07, Volume 2 Number 17 (1999). http://www.intelligententerprise.com/db_area/archives/1999/990712/webhouse.shtml
11. Ralph Kimball, Laura Reeves, Margy Ross, Warren Thornthwaite: The data warehouse lifecycle toolkit. John Wiley & Sons, Inc. ISBN 0-471-25547-5 (1998).
12. Jennifer Widom. Research problems in data warehousing. Proceedings of the Fourth International Conference on Information and Knowledge Management (November 1995).

Change Propagation in an Axiomatic Model of Schema Evolution for Objectbase Management Systems

Randal J. Peters^{*1} and Ken Barker^{**2}

¹ University of Manitoba
Department of Computer Science
Winnipeg, Manitoba, Canada
randal@cs.umanitoba.ca

² University of Calgary
Department of Computer Science
Calgary, Alberta, Canada
barker@cpsc.ucalgary.ca

Abstract. Schema evolution is an important component of advanced information systems such as *objectbase management systems*. These systems typically support volatile and complex application domains that include engineering design, CAD/CAM, multimedia, and geo-information systems. The schema of these applications must be able to evolve along with the changing environment. There are two problems to consider in schema evolution: (i) *semantics of change* and (ii) *change propagation*. The first deals with the effects of the schema change on the overall type system. For example, the deletion of a property in a type affects the subtypes inheriting that property. Our previous work has introduced a sound and complete *axiomatic model* to deal with the semantics of change problem. The second problem deals with the techniques for propagating schema changes to the underlying objects. For example, the addition of an attribute to a type requires additional memory to be allocated to the objects so that values for the attribute may be stored. The first step of change propagation is to identify the affected objects. Subsequent steps carry out the actual changes. This paper deals with the first step by extending the axiomatic model with semantics to determine a sound and complete set of objects affected by a schema change. The extended model can be used with any method for carrying out the changes such as the *conversion*, *screening*, and *filtering* approaches proposed in the literature.

* The Natural Science and Engineering Research Council (NSERC) of Canada support this research under operating grant OGP-0173259.

** The Natural Science and Engineering Research Council (NSERC) of Canada supports this research under research grant RGP-0105566.

1 Introduction

Designers and users of advanced application environments realize the benefits that schema evolution can provide. The main feature is to allow the modification of a design “on the fly.” The main requirement is that there is a clear semantics carried out by each schema change. Support for schema evolution is important when advanced information servers such as *objectbase management systems* (OBMSs) are used to develop and run these applications.

Object-oriented computing is emerging as the predominant technology for providing database services in advanced application domains such as engineering design, CAD/CAM systems, multimedia, and geo-information systems, to name a few. A distinguishing characteristic of these applications is that the schema design can become quite complex with many types and inheritance links between them. An important feature to support in these systems is the ability to modify the schema as the application environment evolves. For example, in an engineering design application many components of an overall design may go through several modifications to produce a final product. Dynamic schema evolution within an OBMS can support these requirements.

Table 1. Typical schema change operations.

	Add (A)	Drop (D)	Modify (M)
Type (T)	Type addition	Type deletion	Add Behavior (AB) Drop Behavior (DB) <div style="border: 1px solid red; display: inline-block; width: 10px; height: 10px; vertical-align: middle;"></div> Add Subtype Relationship (ASR) Drop Subtype Relationship (DSR)

Three basic operations are typically performed during schema evolution: *add*, *drop* and *modify*. Table 1 shows the combinations of applying these operations to types. The result is a collection of six complete operations that are performed on types during schema evolution. We use the conjunction of the abbreviations to denote the operations. For example, AT is an “Add Type” operation and MT-DSR is a “Modify Type - Drop Subtype Relationship” operation. A typical schema change affects many aspects of a system. There are two fundamental problems to consider:

Semantics of Change: The effects of the change on the overall way in which the system organizes information (i.e., the effects on the schema), and

Change Propagation: The effects of the change on the consistency of the underlying objects (i.e., the identification of affected objects and the propagation of the changes to these instances).

For the first problem, the basic approach is to define a number of invariants that must be satisfied by the schema and then define rules and procedures for maintaining these invariants for each possible schema change. Orion [1] and

Gemstone [9] are examples of OBMSs that use this approach. Our approach [12] introduces a *formal axiomatic model* for handling the semantics of change. From the schema designer perspective, it is a simple model to use because the designer only needs to specify and maintain two sets for each type: the *essential supertypes* and *essential properties*. The axioms provide an automated means of managing schema changes based on modifications to these two sets. An important characteristic of the model is that it is proven sound and complete.

For the second problem, the objects affected by a schema change must be identified and then the changes must be carried out. *The main contribution of this paper is an extension to the axiomatic model for identifying objects affected by a schema change.* In keeping with the characteristics of the axiomatic model, this extension has a proven soundness and completeness as well. The result of this work can serve as a “front-end” to any of the proposed techniques for carrying out schema changes. A typical technique is to explicitly *coerce* objects to coincide with the new definition of the schema. *Screening* and *conversion* are two approaches for defining when coercion actually takes place. Conversion (e.g., Orion [1]) stops the system and updates the affected objects immediately after a schema change. Screening (e.g., GemStone [9]) coerces objects when they are first accessed after a schema change (i.e., the system is not stopped to update objects). Sometimes a versioning mechanism is used in conjunction with coercion and old representations of objects are maintained. *Filtering* [14] is a change propagation technique based on a versioning mechanism that maintains older versions of updated objects. The purpose is to provide better compatibility between objects as the schema evolves.

The relationships between the various components regarding the axiomatic model are shown in Figure 1. The *semantics of change* component includes schema modifications by the designer as listed in Table 1 followed by a precise semantics for incorporating these changes at the schema level. This part of the model has been shown to encompass the schema evolution operations of several OBMSs including Orion, Gemstone, O2, and Tigukat. The *change propagation* component identifies the objects affected by the schema change and then carries out the changes by coercing the objects. Object identification is addressed in this paper and can be linked to the various approaches for carrying out changes.

The remainder of the paper is organized as follows. Section 2 gives an overview of the axiomatic object model and its uses in the semantics of change problem of schema evolution. The axiomatic model is extended in Section 3 to develop a change propagation model and form a complete axiomatic model for schema evolution. Other work related to schema evolution and the axiomatic model is outlined in Section 4. Finally, conclusions and future research are given in Section 5.

2 Axiomatic Model Overview

This section gives an overview of an axiomatic model for specifying properties and inheritance structures of types in an object-oriented environment. The model

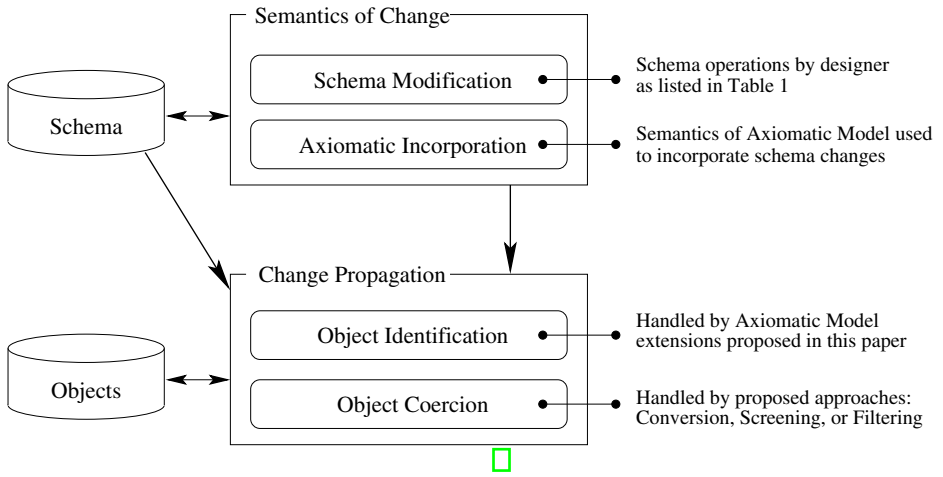


Fig. 1. Schema Evolution Component Relationships of the Axiomatic Model

has been used as a solution to the semantics of change problem in OBMSs [11,12]. It also serves as a basis for a methodology of schema integration in federated objectbase systems [2] and for defining schema evolution in real-time object-oriented database environments [16]. Details of the axiomatic model are given in [11,12]. This section focuses on the notation of the model since they are used in the change propagation extensions presented in Section 3.

A *type* in an object model (called a *class* in some models) defines properties of objects. Existing systems use attributes, methods, and behaviors to represent properties of objects. We use the term *property* to generically encompass all of these components. Types are used as templates for creating objects. The set of all objects created from a particular type is called the *extent* of that type.

Subtyping is a facility of object models that allows types to be built incrementally from other types. We use the symbol “ \preceq ” to represent a reflexive, transitive, and antisymmetric *subtype relationship* where $t \preceq s$ means that type t is a subtype of type s , or equivalently, s is a supertype of t . Diagrammatically, we use a directed arrow from a subtype (the tail) to its supertype (the head) to represent a subtype relationship. A *subtype* inherits all the properties of its supertype and can define additional properties that do not exist in the supertype. If a subtype has multiple supertypes, it inherits the properties of all the supertypes. This is known as *multiple inheritance* and results in a graph of subtype relationships.

A *type lattice* (or simply *lattice*) $L = \langle T, \leq \rangle$ consists of a set of types, T , together with a partial order, \leq , of the elements of T based on the subtype relationship (\preceq). The term *lattice* (or *semi-lattice*) is commonly used in object-oriented literature to denote a typing structure that supports multiple inheritance. This meaning does not correspond to lattice in the strict mathematical sense because the notions of least upper bound and greatest lower bound are relaxed. Regardless, we use the term throughout the paper with the understanding that its object-oriented meaning applies. A type lattice can be represented as a

directed acyclic graph (DAG) with types as vertices and subtype relationships as directed edges.

The notation for the axiomatic model is shown in Table 2. The terms denote various arrangements of types and properties that can be represented in virtually any object model. We address each of these terms and use the simple example type lattice in Figure 2 to clarify their semantics. The example is kept simple so that the functionality of the axiomatic model can be more easily presented and understood. It will become apparent from the following discussion that the model scales up to type lattices of more complex application environments such as those mentioned earlier.

Table 2. Notation for Axiomatic Model

Term	Description
T	The set of all types in an application schema design
L	The type lattice of an application schema design $L = \langle T, \leq \rangle$
s, t, ∇, \perp	Type elements of T
$P(t)$	Immediate supertypes of type t
$P_e(t)$	Essential supertypes of type t
$PL(t)$	All supertypes of type t
L_t	Supertype lattice of type t
$N(t)$	Native properties of type t
$H(t)$	Inherited properties of type t
$N_e(t)$	Essential properties of type t
$I(t)$	Interface of type t
$\alpha_x(f, T^*)$	Apply-all operation

The set of types T represents all the types in an application schema design. These types have schema evolution operations applied to them. The set consisting of all types (i.e., vertices) in Figure 2 forms T in this example. A type lattice L is formed from the set T and the subtype relationships between the types of T . Type elements s and t serve as variables while ∇ and \perp are constants denoting the least defined type and most defined type, respectively. In Figure 2, $\nabla = T_object$ and $\perp = T_null$. Type ∇ serves as a common ancestor of all types and \perp serves as a common descendent. The use of ∇ is popular in many systems as a root with properties that are inherited by all types. For example, it can be used to support object identity or a set of typical comparison operators. The use of \perp , while not as widespread, can be favorable as a type that supports all properties and behaviors. One function is to create a number of “error” objects of this type that can then be returned by the methods of other types when errors occur. These error objects have some meaning with respect to the other methods in the system.

The *immediate supertypes*, $P(t)$, of a type t are those types that cannot be reached from t , transitively, through some other type. In other words, their only link to t is through a *direct* subtype relationship. For example, if we let $t = \mathbf{T_teachingAssistant}$, then the immediate supertypes of t are $\mathbf{T_student}$ and $\mathbf{T_employee}$. Hence, $P(\mathbf{T_teachingAssistant}) = \{\mathbf{T_student}, \mathbf{T_employee}\}$. The other supertypes of $\mathbf{T_teachingAssistant}$ (i.e., $\mathbf{T_person}$, $\mathbf{T_taxSource}$, and $\mathbf{T_object}$) can be reached transitively through $\mathbf{T_student}$ or $\mathbf{T_employee}$.

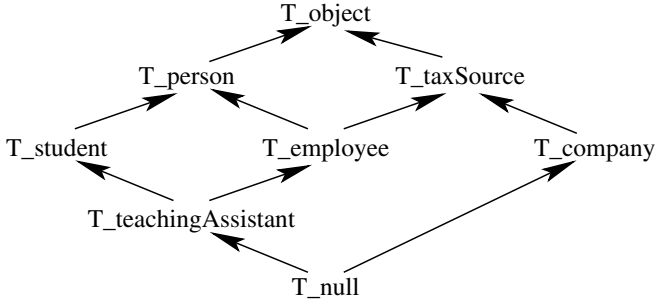


Fig. 2. Simple example type lattice

The *essential supertypes*, $P_e(t)$, are the types identified as being essential to the construction of type t . Essential supertypes must be maintained as supertypes of t for as long as consistently possible during the evolution of the schema. The only way to break a link from t to an essential supertype s is to explicitly remove s from $P_e(t)$ by either dropping the subtype relationship between t and s or by dropping s entirely. Note that $P(t) \subseteq P_e(t)$, which means that the immediate supertypes are essential.

An OBMS can impose constraints that force a newly created type to be a subtype of certain system primitive types. In other words, these primitive types are essential supertypes of every type. For example, many OBMSs define a primitive root type “object” that must be a supertype of all types, either directly or transitively through some other type. Upon creation of a new type t , the system can initialize $P_e(t)$ to $\{\mathbf{T_object}\}$. In a typical environment, the system would provide essential supertypes based on known constraints and the schema designer would provide essential supertypes based on his/her expertise in the particular application domain being modeled.

In Figure 2, assume the system provides the root type $\mathbf{T_object}$ and assume the schema designer has specified the remaining essential supertypes of $\mathbf{T_teachingAssistant}$ resulting in the following:

$$P_e(\mathbf{T_teachingAssistant}) = \{\mathbf{T_student}, \mathbf{T_employee}, \mathbf{T_person}, \mathbf{T_object}\}$$

If $\mathbf{T_student}$ and $\mathbf{T_employee}$ were dropped as immediate supertypes of $\mathbf{T_teachingAssistant}$, then $\mathbf{T_person}$ would be established as an immediate super-

type because it is essential. However, `T_taxSource` would be lost as a supertype because it is not declared as essential.

The *supertype lattice* $L_t = \langle PL(t), \leq_t \rangle$ of a type t consists of a set $PL(t)$ which includes t and all supertypes (immediate, essential, or otherwise) of t together with a partial order \leq_t such that $\forall x, y \in PL(t)$ if $x \preceq y$ in \sqsubseteq then $x \preceq y$ in \leq_t . In other words, if a subtyping relationship exists between supertypes of t in the application lattice, then the subtype relationship also exists in the supertype lattice of t . For example, if we let $t = \text{T_employee}$, then the supertype lattice of t is given as:

$$\begin{aligned} PL(\text{T_employee}) &= \{\text{T_employee}, \text{T_person}, \text{T_taxSource}, \text{T_object}\} \\ \leq_{\text{employee}} &= \{\text{T_employee} \preceq \text{T_person}, \text{T_employee} \preceq \text{T_taxSource}, \\ &\text{T_person} \preceq \text{T_object}, \text{T_taxSource} \preceq \text{T_object}\} \end{aligned}$$

The *native properties*, $N(t)$, of a type t are those properties that are not defined in any of the supertypes of t . That is, they are not inherited from a supertype, but instead are natively defined in t . Note that the native properties of one type may also be defined as native properties of other types that are not in a subtype relationship with one another. For example, the type `T_employee` may have a native “salary” property that is not defined on any of its supertypes. Moreover, `T_person` and `T_taxSource` may both have native “name” properties defined because they are not in a subtype relationship with one another.

The *inherited properties*, $H(t)$, of a type t is the union of the properties defined by all supertypes of t . The native and inherited properties are disjoint. For example, the inherited properties of `T_employee` are the union of the properties defined on `T_person`, `T_taxSource`, and `T_object`. In contrast, the native properties of `T_employee` are those defined on employees, but not defined on any of `T_person`, `T_taxSource`, or `T_object`.

When two common properties are inherited from multiple supertypes (e.g., `T_employee` inherits the “name” property from both `T_person` and `T_taxSource`) a *conflict* can arise and some form of *conflict resolution* must be performed. The conflict resolution problem has been addressed in previous work [12]. One element that must be resolved is the representation/implementation of the property. A simple form of resolution used in some systems is to ask the designer to resolve a conflict by choosing one of the conflicting properties as a basis for the representation/implementation or by redefining the property altogether.

The *interface*, $I(t)$, of a type t is the union of native and inherited properties of t . This term simply serves as a specification of all properties of t to which the object instances of t will respond.

The *essential properties*, $N_e(t)$, are those properties identified as being essential to the construction and existence of type t . Essential properties must be maintained as part of the definition of t for as long as consistently possible during the evolution of the schema. The essential properties of a type consist of all properties natively defined by the type (i.e., $N(t) \subseteq N_e(t)$) and may contain properties inherited from its supertypes. The schema designer has the expertise to understand the properties that types within a particular application domain must support and can declare these properties as being essential to the types by

including them in the appropriate $N_e(t)$ specification. Additionally, the system may require all types to support various primitive properties for object instances such as object identity retrieval and object equality. At type creation time, the system can initialize $N_e(t)$ with the appropriate primitive properties. The synergy between schema designer and system primitives goes hand in hand with the definition of essential properties $N_e(t)$ and essential supertypes $P_e(t)$.

Schema evolution may force inherited properties of a type to be adopted as native properties. This can occur if a type defines an essential property that is currently inherited from a supertype and that property is removed from the supertype or the supertype is removed altogether. For example, assume that a “taxBracket” property is defined on T_taxSource and this property is declared as essential in T_employee. If T_taxSource is deleted, then “taxBracket” would be adopted as a native property of T_employee because it is essential to that type. The axiomatic model automatically handles this adoption process.

We provide an *apply-all* operation in the axiomatic model. This operation, denoted $\alpha_x(f, T^*)$, applies the unary function f to the types $T^* \subseteq T$. The function f is defined over the single variable x , which is shown as the subscript of the α operator. Other variables appearing within the parenthesis of the α operation are substituted with their values prior to evaluation and they remain constant throughout the apply-all operation. The semantics of apply-all will let x range over the elements of T^* and for each type bound to x , f is evaluated and the answer is included in the final result set. If T^* is empty, the empty set is returned. In functional notation, the α operation applies the lambda function $\lambda x.f$ to every element of T^* and returns a set containing the results. For example, the expression $\cup \alpha_x(N_e(x), \{T_{person}, T_{student}\})$ gives the set of native essential properties specified in T_person and T_student.

Table 3 depicts the axioms of dynamic schema evolution using the various types and properties in Table 2. The derivation of the various sets in the axioms are based on the $P_e(t)$ and $N_e(t)$ terms. To define a schema (i.e., type lattice), one need only specify values for these two sets. They can be initialized as part of type creation or modified during schema evolution. All schema operations are handled as modification to these two terms, which eases the burden on the schema designer and makes the system more manageable. The effects of schema changes on subtyping relationships and property inheritance must be closely scrutinized in order to maintain system integrity, as well as the intentions of the schema designer. The axiomatic model provides a consistent, automatic mechanism for deriving the entire type lattice structure after a change to either $P_e(t)$ or $N_e(t)$. Further, the model has proven soundness, completeness, and termination. The axiomatic model has the flexibility to handle variations on type and property arrangements depending on the defaults imposed by individual systems. This results in a powerful model that can be used to describe dynamic schema evolution in OBMSs that support subtyping and property inheritance. The axiomatization and comparison of Tigukat, Orion, GemStone, and O₂ have been addressed in previous work [2,12].

Table 3. Axioms of Dynamic Schema Evolution

Name	Axiom
Axiom of Closure	$\forall t \in T, P_e(t) \subseteq T$
Axiom of Acyclicity	$\forall t \in T, t \notin \cup \alpha_x(PL(x), P(t))$
Axiom of Rootedness	$\exists \nabla \in T, \forall t \in T, \nabla \in PL(t) \wedge P_e(\nabla) = \{\}$
Axiom of Pointedness	$\exists \perp \in T, \forall t \in T, t \in PL(\perp)$
Axiom of Direct Supertypes	$\forall t \in T, P(t) = P_e(t) - \cup \alpha_x(PL(x) \cap P_e(t) - \{x\}, P_e(t))$
Axiom of Supertype Lattice	$\forall t \in T, PL(t) = \cup \alpha_x(PL(x), P(t)) \cup \{t\}$
Axiom of Interface	$\forall t \in T, I(t) = N(t) \cup H(t)$
Axiom of Nativeness	$\forall t \in T, N(t) = N_e(t) - H(t)$
Axiom of Inheritance	$\forall t \in T, H(t) = \cup \alpha_x(I(x), P(t))$

The specification and management of P_e and N_e can be a shared responsibility between the system and the user. For example, when a new type is defined, the system may open a dialog with the schema designer to determine all supertypes and properties that are essential to the new type. Alternatively, the system may make a default assumption that all supertypes and properties (including inherited properties) are essential in a given type. Other configurations are possible as well. Current systems vary in the semantics defined for the notions of subtyping, inheritance, and nativeness. The formalization of these concepts into the axiomatic model gives a common basis that allows systems the flexibility to build their own customized notions on top of them, while remaining rooted at the formal model. The flexibility of the axiomatic model has been shown by extending it to support schema integration [2] and real-time database systems [16]. □

3 Change Propagation Model

This section develops the change propagation extensions to the axiomatic model described in Section 2. Additional notation is introduced along with a list of new axioms to support consistent change propagation. The first extension introduces a new notation for identifying the objects in the extent of the types. For each type t , there are three sets of objects associated with t denoted $E_0(t)$, $E_\infty(t)$, and $E_i(t)$. These sets correspond to the shallow extent, deep extent, and i^{th} level extent, respectively. The shallow extent of a type t is the set of objects created directly from t . The deep extent of t includes its shallow extent along with the shallow extent of all subtypes of t . The i^{th} level extent of t includes its shallow extent and the shallow extent of the subtypes up to depth i . Shallow and deep extents are well known concepts and are used extensively in OBMSs for object query models and query processing. The i^{th} level extent is included for completeness as it is useful in some aspects of objectbase systems [13].

Changes to the schema affect the structure of the types and the organization of the type lattice. This in turn affects the objects in the type extents. The first step in performing change propagation is to identify the affected objects. The subsequent steps carry out the changes to the objects by coercing them into structures that correspond to the evolved types. The literature reports three basic approaches for object coercion: *conversion* [9], *screening* [1], and *filtering* [14]. Conversion stops the system and updates the affected objects immediately after a schema change. This is a straightforward and simple approach, but can suffer from performance problems if many changes occur and the system is halted frequently. Screening coerces objects when they are first accessed after a schema change. The system does not have to be stopped to update objects and so concurrent operations can proceed with greater transparency. Filtering is based on a versioning mechanism that maintains older versions of updated objects. The approach requires more space overhead and greater processing demands since translations between older and newer versions may be required on a continuous basis. The purpose is to provide better compatibility between objects and their method implementations as the schema evolves.

The change propagation model presented in this paper is the first of its kind that clearly and formally identifies the objects affected by a schema change. The model can be adopted as a “front end” to any of the coercion approaches. The model is *complete* in the sense that it guarantees to determine all objects affected by a schema change. This is a minimal requirement of any change propagation mechanism. The model is *sound* in the sense that it only determines objects that are guaranteed to be affected by a schema change. In other words, it guarantees a minimal set of objects affected by a schema change. This translates into greater efficiency because some objects of certain subtypes may not have to be coerced. Other approaches are more liberal and may coerce objects that did not require an update. For example, if a type t is affected by a schema change, some systems simply coerce all the objects of t and its subtypes (i.e., the deep extent of t). There are cases where many objects in the deep extent are not affected by the change. Our model will automatically exclude these from being coerced. Another property of the model is that it only uses schema information to determine the objects affected by a schema change. It does not have to access the objects themselves. Finally, the model provides a precise semantics for identifying the objects affected in change propagation. Other schema evolution approaches give informal explanations of the change propagation operations. The axiomatic model can provide a formal basis for comparing the various techniques.

Two theorems regarding the completeness and soundness of the change propagation axioms are constructed. Only sketches of the proofs are given. The complete proofs are omitted due to page limitations.

Theorem 1. *The change propagation axioms are complete.*

Proof. By induction on *maximal paths* from the types affected by the schema evolution operation. A *maximal path* is the largest number of direct supertype links between two types. For example, in Figure 2 the maximal path between

$T_{\text{teachingAssistant}}$ and T_{person} is two, and the maximal path between T_{null} and $T_{\text{taxSource}}$ is three. The proof proceeds by showing the completeness of each axiom in Table 4. The base cases for the induction focus on the types involved as parameters of the schema change operation. The induction step is straightforward and shows that given the assumption that the axioms are complete for types with maximal path n , the axioms are complete for types with maximal path $n+1$. \square

Theorem 2. *The change propagation axioms are sound.*

Proof. By induction on maximal paths from the types involved in the schema evolution operation. The proof is constructed similar to Theorem 1 above. \square

Table 4. Notation for Axiomatic Model

Axiom Name	Axiom
Add Behavior	$MT - AB(t, b) = \cup \alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge b \notin I(r)\})$
Drop Behavior	$MT - DB(t, b) = \cup \alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge \neg \exists v(v \in PL(r) - \{t\} \wedge b \in N_e(v))\})$
Add Subtype Relationship	$MT - ASR(t, s) = \cup \alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge I(s) \neq \emptyset \wedge I(s) \neq (I(s) \cap I(r))\})$
Drop Subtype Relationship	$MT - DSR(t, s) = \cup \alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge \neg \exists v(v \in PL(r) - \{t\} \wedge s \in P_e(v)) \wedge I(s) \neq \emptyset \wedge I(s) \cap \cup \alpha_y(N_e(y), PL(r) - \{s\})\})$
Add Type	$AT(t, \{s_1, \dots, s_n\}, \{r_1, \dots, r_m\}, \{b_1, \dots, b_j\}) = \cup \alpha_y(\cup \alpha_x(MT - ASR(x, y), \{r_1, \dots, r_m\}), \{t\} \cup \{s_1 \dots s_n\})$
Drop Type	$DT(t) = \cup \alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge I(t) \neq \emptyset \wedge I(t) \neq I(t) \cap \cup \alpha_y(N_e(y), PL(r) - \{t\})\})$

The axioms of change propagation are shown in Table 4. They determine the sound and complete set of objects affected by the schema change operations on types outlined in Table 1. The purpose of each axiom is to return a sound and complete set of objects affected by the schema change associated with the axiom.

The semantics of each axiom is explained below along with an example of its action. Refer to Figure 2 for the type lattice structure of the examples and Table 5 for the essential properties and essential supertypes of each type in the example. All examples below use Figure 2 and Table 5 as a starting point - the examples are not cumulative.

Add Behavior (MT-AB): Adds behavior b as an essential property of type t (i.e., $N_e(t) = N_e(t) \cup \{b\}$). If b was not part of t then the objects of t must now support the new behavior b . This may require an update to the objects of t . For example, if b is implemented as a stored attribute then the objects of t

Table 5. Example Types

Type	Essential Supertypes (P_e)
T_object	
T_person	T_object
T_taxSource	T_object
T_student	T_person, T_object
T_employee	T_person, T_taxSource, T_object
T_company	T_taxSource, T_object
T_teachingAssistant	T_student, T_employee, T_person, T_object

Table 6. Example Properties

Type	Essential Properties (N_e)
T_object	
T_person	Name, Age
T_taxSource	Name, TaxBracket, GovID
T_student	Name, Grade
T_employee	Name, Salary, Age, TaxBracket
T_company	Name, Revenue, Phone
T_teachingAssistant	Hours, TaxBracket, GovID, Salary, Age

require additional memory to store the value of the attribute. Furthermore, the subtypes of t could inherit b as a new behavior and so their objects may be affected. The axiom determines all the types (r) that are a subtype of t and do not have b as part of their interface. The extended union over the shallow extent of these types gives the set of objects affected by the schema change.

Example: Suppose a Government ID (GovID) property is added to type T_person for identification purposes. This would add GovID to $N_e(T_{person})$. Since this is a new property for T_person, the objects in the shallow extent of T_person are affected (i.e., $E_0(T_{person})$). Furthermore, the objects in the shallow extent of T_student are also affected, but not the objects in the deep extent of T_employee because T_employee inherits the GovID property from T_taxSource. It may appear that a conflict has arisen for GovID between types T_person and T_taxSource. This can be averted because the representation/implementation of GovID in T_employee has been previously decided by its prior subtyping relationship with T_taxSource.

Drop Behavior (MT-DB): Deletes behavior b from the essential properties of type t (i.e., $N_e(t) = N_e(t) - \{b\}$). This will affect the objects of t only if b is natively defined on t (i.e., t does not inherit b from some other type). This may also affect the subtypes of t that only inherit b from t . The variable r is used to build a set of types that consist of the subtypes of t that inherit b only from

t. This is accomplished by specifying there does not exist a *v* such that *v* is a supertype of *r* (not including *t*) and *b* is an essential behavior of *v*. The union over the shallow extent of these types gives the set of objects affected by the schema change.

Example: If a schema operation drops property Age from the essential properties of T_person, then the objects in the shallow extent of T_person and T_student are affected. However, the deep extent of T_employee is not affected because it declares Age as an essential property that now becomes native to T_employee, so the interface of T_employee is not affected.

Add Subtype Relationship (MT-ASR): Adds a new subtype relationship between type *t* and type *s* ($t \leq s$) by adding *s* as an essential supertype of *t* (i.e., $P_e(t) = P_e(t) \cup \{s\}$). The type *t* and its subtypes can be affected by this change if *s* introduces new properties that are inherited. The objects of a subtype *r* are affected by the schema change if the interface of *s* is not empty and it is not a subset of the interface of *r*. Again, the union over the shallow extent of these types gives the set of objects to be coerced.

Example: If T_taxSource is added as an essential supertype of T_student, then the objects in the shallow extent of T_student must have changes propagated to them because of the newly inherited behaviors taxBracket and GovID. However, the objects in the deep extent of T_teachingAssistant are not affected because this type has another link (or path) to T_taxSource through T_employee.

Drop Subtype Relationship (MT-DSR): Drops an existing subtype relationship between type *t* and type *s* by removing *s* as an essential supertype of *t* (i.e., $P_e(t) = P_e(t) - \{s\}$). The type *t* and its subtypes can be affected by this change if they inherit some behaviors only from *s* and all links to *s* are lost by dropping the subtype relationship from *t*. The variable *v* is used to determine that there are no other subtype links to *s*. The second line of the axiom determines if the behaviors of *s* are inherited from some other type even if all the subtype links to *s* are lost.

Example: Suppose the subtype relationship from T_teachingAssistant to T_employee is dropped. Now, T_teachingAssistant has no other link to T_employee or T_taxSource. However, the objects in its deep extent are not affected because T_teachingAssistant has declared a set of essential properties that include all the properties that it previously inherited from T_employee and T_taxSource. Since the properties are essential, they are kept with T_teachingAssistant. Thus, no objects are affected by change propagation in this case.

Add Type (AT): Add the type *t* to the schema with $\{s_1, \dots, s_n\}$ as the essential supertypes of *t*, $\{r_1, \dots, r_m\}$ as the initial subtypes of *t*, and $\{b_1, \dots, b_j\}$ as the essential properties of *t*. Subtypes $\{r_1, \dots, r_m\}$ can be incorporated by adding a new subtype relationship from each *r_i* to the new type *t*. This change can affect the objects in $\{r_1, \dots, r_m\}$ and their subtypes because they may now inherit some properties in $\{b_1, \dots, b_j\}$. Furthermore, the new type *t* may transitively introduce new subtyping relationships between the types $\{r_1, \dots, r_m\}$ and the types $\{s_1, \dots, s_n\}$. This may result in some of the properties defined on

$\{s_1, \dots, s_n\}$ being inherited by $\{r_1, \dots, r_m\}$, thus affecting the objects in their extents.

Example: Suppose a new type T_worker is added with essential supertypes $\{T_object, T_taxSource\}$, initial subtype $\{T_employee\}$, and essential properties $\{GovID, WorkersCompID\}$. Due to the axiom of rootedness and axiom of pointedness in Table 3, the system can automatically include T_object as an essential supertype and T_null as an initial subtype. The direct supertype links in the lattice will change as shown in Figure 3 so that there is a direct link from $T_employee$ to T_worker and a direct link from T_worker to $T_taxSource$. The direct link from $T_employee$ to $T_taxSource$ is lost. The new property $WorkersCompID$ (representing a worker's compensation identifier) is inherited by $T_employee$ and $T_teachingAssistant$ and so the objects of these two types are affected by the schema change.

The following example shows how a newly added type can affect the initial subtypes of the new type by transitively inheriting behaviors of the essential supertypes.

Example: Consider the generic type lattice in Figure 4(a) and the schema operation $AT(t, \{s_1\}, \{r_1, r_2\}, \{p_z\})$ that adds type t with essential supertype $\{s_1\}$, initial subtypes $\{r_1, r_2\}$, and essential property $\{p_z\}$. The essential properties of each type in the figure are listed below the type. Clearly, type r_1 is affected by the addition because it will inherit property p_z from the new type t . It appears that r_2 should not be affected because it already defines property p_z . However, r_2 is affected because it transitively inherits property p_x from s_1 through t . Note that r_1 also transitively inherits p_x . The updated lattice is shown in Figure 4(b).

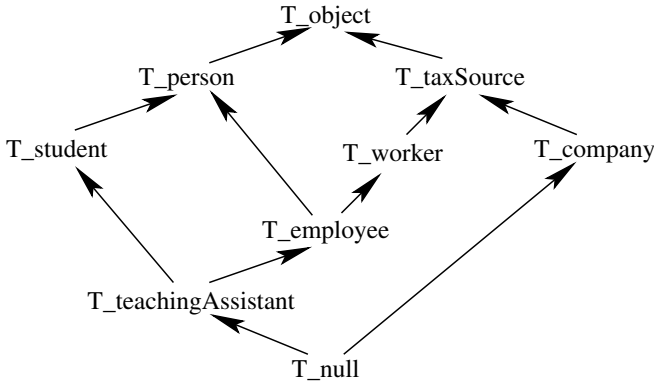


Fig. 3. Revised type lattice with type T_worker added

Drop Type (DT): Drops the type t from the lattice. This axiom is a simplified version of the MT-DSR axiom. It is simpler because there is no need

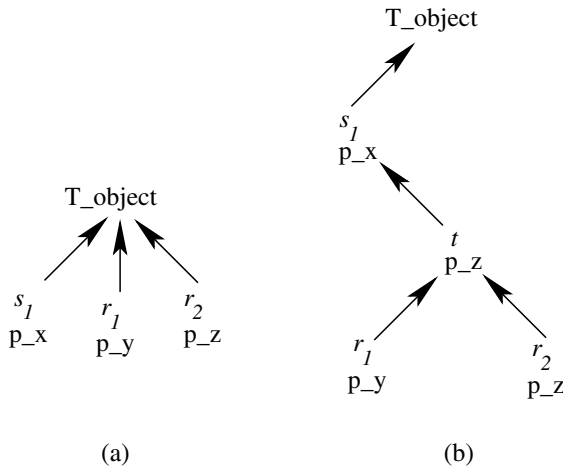


Fig. 4. Generic type lattice illustrating transitive inheritance for Add Type operation

to check for other subtype relationships to t since t will be removed from the lattice.

Example: If $T_employee$ is dropped from the lattice, then the link from $T_teachingAssistant$ to $T_taxSource$ is lost because $T_taxSource$ is not an essential supertype of $T_teachingAssistant$. This means that $T_teachingAssistant$ will no longer inherit the properties of $T_taxSource$. Regardless, the objects in the deep extent of $T_teachingAssistant$ are not affected by the schema change because all the properties defined by $T_employee$ and $T_taxSource$ have been defined as essential properties of $T_teachingAssistant$, or they are inherited from T_person .

If $T_student$ is now dropped from the lattice, then a direct link from $T_teachingAssistant$ to T_person is established because T_person is an essential supertype of $T_teachingAssistant$. The objects in the deep extent of $T_teachingAssistant$ are affected because the `Grade` property defined by $T_student$ is not defined as an essential property of $T_teachingAssistant$ and so it is lost.

The axioms of Table 4 give a precise semantics for determining a sound and complete set of objects affected by schema evolution operations on types. These objects require a coercion mechanism to be applied (e.g., conversion, screening, filtering) so that they correspond to the updated version of the schema. It is clear that the axiomatic model can be used as a formal underlying model of a complete schema evolution mechanism for object-oriented environments. The descriptions and examples following the axiom specifications are not meant to provide a complete explanation. Many subtle qualities can be observed when using the axioms in various examples that are too great in number to present in the limited space allowed for this paper. We have implemented a tool for defining schema and performing schema evolution based solely on essential properties and essential supertypes. The tool currently supports all the axioms for the semantics

of change problem. The tool is currently being updated to incorporate the axioms for change propagation along with user selectable options for determining which coercion mechanism to apply to affected objects.

4 Related Work

Various systems have proposed solutions to the problems of *semantics of change* and *change propagation* for schema evolution in OBMSs. To support semantics of change, the most common approach is to define a number of invariants that must be satisfied by the schema, along with a set of rules and procedures that maintain the invariants with each schema change. To support change propagation, one solution is to explicitly *coerce* objects to coincide with the new definition of the schema. This technique updates the affected objects, changing their representation as dictated by the new schema. Unless a versioning mechanism is used in conjunction with coercion, the old representations of the objects are lost. *Screening*, *conversion*, and *filtering* are techniques that define when and how coercion takes place.

In *screening*, schema changes generate a conversion program that is independently capable of converting objects into the new representation. The coercion is not immediate, but rather is delayed until an instance of the modified schema is accessed. That is, object access is monitored by the system and whenever an outdated object is accessed, the system invokes the conversion program to coerce the object into the newer definition. Conversion programs resulting from multiple independent changes to a type are composed; meaning access to an object may invoke the execution of multiple conversion programs where each one handles a particular change to the schema. Screening causes processing delays during object access because the conversion program may have to be applied. Furthermore, it can be difficult to determine when the system no longer needs to check whether a particular conversion program is required. This can cause overhead during every object access and may increase the amount of supplementary information that the system needs to keep in the form of screening flags.

In *conversion*, each schema change initiates an immediate coercion of all objects affected by the change. This approach causes processing delays during schema modifications, but delays are not incurred during object access. Once conversion is complete, all objects are up to date.

Another solution for handling change consistency of instances is to introduce a new version of the schema with every modification and supplement each schema version with additional definitions that handle the semantic differences between versions. These additional definitions are known as *filters* and the technique is called *filtering*. Error handlers are one example of filters. They can be defined on each version of the schema to trap inconsistent access and produce error and warning messages.

In the filtering approach, changes are never propagated to the instances. Instead, objects become instances of particular versions of the schema. When the schema is changed, the old objects remain with the old version of the schema and

new objects are created as instances of the new schema. The filters define the consistency between the old and new versions of schema and handle the problems associated with properties written according to one version accessing objects of a different version. This approach introduces the overhead of maintaining the separate versions and the filters between them that need to be applied from time to time.

A *hybrid* approach combines two or more of the above methods. For example, a system could use filtering as the underlying mechanism and allow explicit coercion to newer versions of types through screening or conversion. This could be used to reduce the overhead in the number of versions and filters that need to be maintained. Another example is a system that takes an active role by using screening as the default and switching to conversion whenever the system is idle.

The axiomatic model of change propagation is responsible for identifying the sound and complete set of objects affected by a schema change. This set can serve as input to any of the coercion methods described above. Thus, the axiomatic model can act as a “front-end” to systems using these approaches.

Orion [1,5] is the first system to introduce an invariants and rules approach as a structured way of describing schema evolution in OBMSs. Invariants define the consistency of the schema under the constraints of the object model. Rules are introduced to guide the preservation of the invariants when choices in modifying the schema arise. Orion defines five invariants and a set of twelve accompanying rules for maintaining the invariants over schema changes. Orion’s taxonomy of changes represents the majority of typical schema modifications allowed in most OBMSs. Change propagation in Orion is handled through screening that coerces out-of-date objects to new schema definitions when the objects are accessed.

Schema evolution in GemStone [9] is similar to Orion in its definition of a number of invariants. The GemStone model is less complex than Orion in that multiple inheritance and explicit deletion of objects are not permitted. As a result, the schema evolution policies in GemStone are simpler and cleaner, but not as powerful as those of Orion. For example, while Orion defines twelve rules to clarify the effects of schema modification, GemStone requires no such rules. Conversion is used in GemStone to propagate changes to the instances. Literature on GemStone mentions the possibility of a hybrid approach that allows both conversion and screening, but it is not clear if such a system has been developed.

Skarra and Zdonik [14,15] define a framework for versioning types in the Encore object model as a support mechanism for evolving type definitions. Their work is focused on dealing with change propagation. The schema evolution operations of Encore are similar to Orion. The authors introduce a generic type as a collection of individual versions of that type. This is known as the *version set* of the type. Every change to a type results in the generation of a new version of that type. Since a change to a type can also affect its subtypes, new versions of the subtypes may also be generated. By default, objects are bound to a specific type version and must be explicitly coerced to a newer version. Since objects are bound to a specific type version, a problem of missing information can arise if programs (i.e., methods) written according to one type version are applied

to objects of a different version. For example, if a property is dropped from a type, programs written according to an older type version may no longer work on objects created with the newer version because the newer object is missing some information (i.e., the dropped property). Similarly, if a property is added to a type, programs written with the newer type version in mind may not work on older objects because of missing information. For this reason, type versions include additional definitions, called *handlers*, which act as filters for managing the semantic differences between versions. This approach is the first to address the issue of maintaining consistency between versions of types.

Nguyen and Rieu [7] discuss schema evolution in the Sherpa model and compare their work to Encore, Gemstone, Orion, and one of their earlier models for CAD systems called Cadb. The emphasis of this work is to provide equal support for semantics of change and change propagation. The schema changes allowed in Sherpa follow those of Orion. Schema changes are propagated to instances through conversion or screening, which is selected by the user. However, only the conversion approach is discussed. Change propagation is assisted by the notion of *relevant classes*. A *relevant class* is a semantically consistent partial definition of a *complete class* and is bound to the class. A relevant class is similar to a type version in [14] and a complete class resembles a version set. The purpose of relevant classes is to evaluate the side effects of propagating schema changes to the instances and to guide this propagation.

In OTGen [6] the focus shifts from dynamic schema evolution to database reorganization. The invariants and rules approach is used, and the typical schema changes are allowed. The invariants are used to define default transformations for each schema change. Schema changes produce a transformation table that describes how to modify affected instances. Multiple schema changes are usually grouped and released as a package called a *transformer*. Screening is used to apply the transformer and propagate changes to the instances. Multiple releases are composed and, thus, access to an older object can invoke multiple transformers to bring the object up to date. One result of the database reorganization approach is that multiple changes are packaged into a single release and this is expected to reduce the number of screening operations that need to be invoked for each object access. Another result is that transformers are represented as tables that are initialized by OTGen.

The Tigukat OBMS [8] incorporates a uniform object model and schema evolution is handled as type and property extensions to the base model. As discussed in Section 2, a sound and complete axiomatic model for the semantics of change problem has been developed in the context of Tigukat. This axiomatic model is used to compare the schema evolution operations of Tigukat, Orion, GemStone, O2, and others [2,12]. Change propagation in Tigukat is handled by a filtering approach that uses behavior histories [10], which are based on the temporal aspects of the object model [3,4]. When a change is made to the schema, the change is not automatically propagated to the instances. Instead, the old version of the schema is maintained and the change is recorded in the proper temporal histories. Existing objects continue to maintain the characteristics of

the older schema while newly created objects correspond to the semantics of the newer schema. Coercion of older objects to newer versions of the schema is optional in Tigukat. Since different versions of types are maintained through temporal histories, the schema information of older objects is available and can be used to continue processing these objects in a historical manner. If coercion is desired, the entire object does not need to be updated at once. Objects can be coerced to a newer version of the schema one property at a time. This means that some properties of an object may work with newer versions, while others may work with older ones. This is in contrast to other models where an object is converted in its entirety to a newer schema version, thereby losing the old information of the object.

5 Conclusion

The formal axiomatic model of schema evolution is a powerful mechanism for reasoning about objectbase management systems. The model consists of two major components: namely, (i) *semantics of change*, and (ii) *change propagation*. Our previous work [12] deals with the first component. This paper extends that work to change propagation. The first step in performing change propagation is to identify the affected objects. The subsequent steps carry out the changes to the objects by coercing them into structures that correspond to the evolved types. The axiomatic model extensions described in this paper give a precise semantics for identifying the objects affected by a schema change. The extended model determines the affected objects using existing information from the type system without making any changes to the type system itself. This could be used in conjunction with a graphical display to show the implications of a schema change and allow the designer to easily cancel the change. We show that this model can be used as a “front-end” component to various coercion methods. Further, the model ties in with the axiomatic model for the semantics of change problem [12] to form a complete axiomatic model of schema evolution in OBMSs. The model is easy to work with because the designer only needs to specify and maintain two sets for each type: the *essential supertypes* (P_e) and *essential properties* (N_e). Subsequently, the axioms provide an automated means of managing the *semantics of change* and *change propagation* based on modifications to these two sets. The utility of this model is further demonstrated by its inherent automation. Application of the axioms is all that is required to determine which objects are affected. Other researchers have demonstrated the utility of the axiomatic model by extending it into areas such as real-time systems Zhou *et al.* [16], but their research does not address the issue of change propagation. Finally, the axiomatic model is proven sound and complete by the work in this paper together with previous work [12].

This research can be extended in several directions. Our current schema management tool implements the *semantics of change* system [12]. Clearly, the change propagation described in this paper can be added to the schema management tool to enhance its functionality. Another interesting research direc-

tion is to determine how these techniques can be applied to a federated system that requires integration. For example, adding a “new” database into a federated schema could be seen as little more than a change to the existing integrated schema. Successful application of these techniques would make substantial progress in automating the process of system integration. Another research direction to consider is extending the model to manage security within complex systems such as OBMSs. An axiomatic representation of role based security models may assist in the architecture and management of changing security permissions in these systems.

Another problem to investigate is the extension into object view models and the management of object view schemas. Evolution of view schemas is more challenging because different perspectives are placed on objects according to the groups of users accessing them. In addition, closure constraints must be taken into consideration when developing view schemas. Closure refers to the condition where including a certain type (or property) in a view, requires other types (and/or properties) be added to ensure the new ones are meaningful. For example, if you create a view that has a *T_car* type with a *Manufacturer* property, then the *T_company* type should be included in some form so that the *Manufacturer* can be related to it. One open question asks what portions of *T_company* should be included to be semantically meaningful without compromising security or scope considerations. Another question asks to what level should the inclusion be carried out. That is, if *T_company* is included with some properties then what other types need to be included for them to be meaningful?

References

1. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth, “Semantics and Implementation of Schema Evolution in Object-Oriented Databases,” in *Proc. ACM SIGMOD Int’l Conf. on Management of Data*, 1987, pp. 311-322.
2. A. A. D’Silva, “Dynamic Evolution of Integrated Schemas for Federated Objectbase Systems.” University of Manitoba, Computer Science, M.Sc. 1998.
3. I. Goralwalla, D. Szafron, M. T. Özsu, and R. J. Peters, “Managing Schema Evolution using a Temporal Object Model,” in *Proc. Int’l Conf. on Conceptual Modeling*, 1997.
4. I. Goralwalla, D. Szafron, M. T. Özsu, and R. J. Peters, “Approach to Managing Schema Evolution in Object Database Systems,” *Data & Knowledge Engineering*, 1998. In press - to appear November 1998.
5. W. Kim and H.-T. Chou, “Versions of Schema for Object-Oriented Databases,” in *Proc. Int’l Conf. on Very Large Databases*, 1988, pp. 148-159.
6. B. S. Lerner and A. N. Habermann, “Beyond Schema Evolution to Database Reorganization,” in *Proc. Int’l Conf. on Object-Oriented Programming, Languages, Applications*, 1990, pp. 67-76.
7. G. T. Nguyen and D. Rieu, “Expert Database Support for Consistent Dynamic Objects,” in *Proc. Int’l Conf. on Very Large Databases*, 1987, pp. 493-500.
8. M. T. Özsu, R. Peters, D. Szafron, B. Irani, A. Lipka, and A. Muñoz, “TIGUKAT: A Uniform Behavioral Objectbase Management System,” *The VLDB Journal*, 4(3):445-492, 1995. Special issue on persistent object systems.

9. D. J. Penney and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS," in Proc. Int'l Conf. on Object-Oriented Programming, Languages, Applications, 1987, pp. 111-117.
10. R. J. Peters, "TIGUKAT: A Uniform Behavioral Objectbase Management System." University of Alberta, Computing Science, Ph.D. 1994. Available as University of Alberta Technical Report TR94-06.
11. R. J. Peters and M. T. Özsu, "Axiomatization of Dynamic Schema Evolution in Objectbases," in Int'l Conf. on Data Engineering. Taiwan, 1995, pp. 156-164.
12. R. J. Peters and M. T. Özsu, "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems," ACM Transactions on Database Systems, 22(1):75-114, 1997.
13. G. Shaw and S. Zdonik, "A Query Algebra for Object-Oriented Databases," in Proc. Int'l Conf. on Data Engineering, 1990, pp. 154-162.
14. A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," in Proc. Int'l Conf. on Object-Oriented Programming, Languages, Applications, 1986, pp. 483-495.
15. A. H. Skarra and S. B. Zdonik, "Type Evolution in an Object-Oriented Database," in Research Directions in Object-Oriented Programming: MIT Press, 1987, pp. 393-415.
16. L. Zhou, E. A. Rundensteiner, and K. G. Shin, "Schema Evolution of an Object-Oriented Real-Time Database System for Manufacturing Automation," IEEE Trans. on Knowledge and Data Eng., 9(6):956-977, 1997.

Evolving Objects: Conceptual Description of Adaptive Information Systems^{*}

Gunter Saake¹, Can Türker², and Stefan Conrad³

¹ Otto-von-Guericke-Universität Magdeburg
Institut für Technische und Betriebliche Informationssysteme
Postfach 4120, D-39016 Magdeburg, Germany
`saake@iti.cs.uni-magdeburg.de`

² Swiss Federal Institute of Technology (ETH) Zurich
Institute of Information Systems, ETH Zentrum
CH-8092 Zurich, Switzerland
`tuerker@inf.ethz.ch`

³ Ludwig-Maximilians-Universität München
Institut für Informatik
Oettingenstr. 67, D-80538 München, Germany
`conrad@dbis.informatik.uni-muenchen.de`

Abstract. Today, information systems are essential parts of large organizations. Since such kinds of systems have a very long life-span, they have to be adapted to new changing requirements occurring during their lifetime. Evolution must be regarded not only at the object state level, but also at the object behavior level. Especially, the explicit handling of (behavior) evolution on the conceptual level is necessary. For that, we introduce the notion of *evolving* objects as basic building blocks of information systems. The behavior of such an object is divided into a *rigid* and an *evolving* part. The rigid behavior is ideally stable for the whole life-span of the object; the evolving behavior can be changed dynamically at runtime. In this paper, we present an extended specification framework for modeling evolving objects. Particularly, this framework provides the basis to explicitly specify behavior evolution.

Keywords: Evolving objects, behavior evolution, adaptive information systems, object specification.

1 Introduction

The development and maintenance of information systems is a particular important task, because information systems often provide the software infrastructure within companies. Following the phases of the software life cycle, the development of an information system starts with a requirements analysis phase, followed by conceptual design, implementation and testing. Finally, the system is

^{*} This research was partially supported by the ESPRIT Basic Research Working Groups ASPIRE (No. 22704) and FireWorks (No. 23531) and by the German Science Foundation DFG (SPP 1064: SA 465/19).

put into operation. In practice, a running system is subject to ongoing construction. On the one hand, improvements on the implementation level, for instance removing bugs or introducing more efficient data structures, have to be carried out. On the other hand, the hardware as well as the underlying software like the operating system can change. Often this leads to adaptations in the implementation.

Unfortunately, there also are changes occurring on the requirements and conceptual level. Business rules may change in the course of time, laws may change and, thereby, require that information systems have to be adapted in order to follow the new business rules or laws. To get a grasp of changing requirements, flexible approaches are needed which allow to describe *evolving* requirements. Due to the fact that information systems usually consist of large numbers of long-living objects, the objects themselves must be able to evolve. In this context, evolution does not only refer to the change of object states but also to the change of the object behavior, i.e., the rules (or axioms) which describe the allowed dynamic behavior of objects may change during the existence of the affected objects.

Considering the current state-of-the-art in the area of requirements specification and conceptual modeling, we have to face the fact that none of the currently known approaches is able to adequately capture this problem of changing requirements. Of course, if we know in advance all possible changes which might occur eventually, we can directly code them into the specification of the system. Unfortunately, we in general know only some changes possibly occurring in the future. In practice, unforeseen changes of requirements occur quite frequently and must be respected in existing and running information systems.

Neither well-known approaches to conceptually describing information systems, like OMT [18], OOA&OOD [2] and UML [3], nor formal specification approaches, like TROLL [13,19], TROLL *light* [11,12], Albert [9], or CMSL/LCM [24], provide real support for dealing with changing requirements. Whereas the meaning of a traditional specification is clear, the meaning of changes on the conceptual or specification level is often not obvious. A (traditional) specification describes possible life cycles of an information system (which, for instance, could be expressed by means of states and state transitions). Changing requirements during runtime of a system does mean that we have to change the specification of the system, e.g., we add new rules (axioms), remove existing ones, or change them. For instance, if we remove a rule from the specification of a system, this rule is valid for the part of the life time of the system before that change, and it is not valid afterwards.

The remainder of this paper is organized as follows. We start with a comparison of program development and development of information systems. Coming from the well-known metaphor of “*program as house*” which characterizes the traditional development of programs, we introduce a new metaphor for the development of information systems: the “*information system as city*” metaphor. In Section 3 we consider the requirements for changes (or *adaptations*) of database

applications and briefly sketch the possibilities being currently available in relational database systems for changing the behavior of the system during runtime.

Although these requirements demand for a continuous engineering [16] of database and information systems, on a conceptual level there is no real support for that. The current modeling and specification technology (in Section 4 we briefly sketch the state-of-the-art in the object specifications area [\[4\]](#), considering the object specification TROLL as a representative example) does not allow to model or specify changes which might occur during runtime of a system in a flexible way. In order to overcome this limitation, we develop an approach for modeling and specifying *evolving objects*, i.e., objects in an information system for which integrity constraints and the behavior rules (axioms) may change during the existence of the object (see Section 5). This seems to be a major step towards supporting continuous engineering of information systems on the conceptual or modeling level because all traditional approaches require the behavior of objects to be completely fixed at specification time.

2 Programs versus Information Systems

A common metaphor in software engineering is to see program construction as similar to building a house: an architect is planning the structure, building a house is a sequence of concrete steps, and the building process terminates after finishing the house. In this section we will discuss whether this metaphor can be applied to information systems as well or has to be adapted to new requirements.

2.1 ‘Program-as-House’ Metaphor

This metaphor can be characterized by some observations which can be transferred from house-building to software construction:

- House-building is primarily concerned with manufacturing *one* building resulting in a high coherence of parts and steps. It is possible to build one house under supervision of only one architect, too.
- In terms of computer science, the building process resembles a deterministic sequence of steps and may even be seen as an ‘algorithm’. The building process consists of concrete steps following more or less a determined schedule (in practice, however, this process is not so smooth anyway). As an important aspect there is a *termination* of the building process when the construction is finished.

The main part of existing software engineering methods is devoted to this process of constructing a new ‘software building’. Practical computing infrastructures, however, show problems which can be more adequately described by other metaphors.

2.2 ‘Information-System-as-City’ Metaphor

For information systems, the ‘information-system-as-city’ metaphor seems to be more appropriate: an information system consists of many buildings (=programs) using a shared infrastructure; building a city is a vivid and sometimes chaotic process; there are rather restrictions than concrete prescriptions for building houses; and the construction process will never be finished. Old and new buildings have to co-exist, and old buildings are sometimes used for purposes they have never been intended for.

This metaphor can be characterized as follows:

- In a city there are thousands of buildings but one infrastructure connecting the separate buildings. This infrastructure is the essence which keeps a city running: transportation, electricity, telecommunication, public services. Due to this complexity there have to be several architects which guide the city development.
- In a city, the building process can be seen as a ‘living system’ rather than a prescribed process. At each time, parallel, unrelated steps are performed on sometimes independent, but often correlated construction places. The building process is characterized by restrictions rather than by prescriptions and leaves some freedom to the local architects. A city may never reach a final state (or it is dead). In a living city, old and new buildings co-exist — not always in harmony but in some cooperation.
- Building a completely new city is an event as rare in history as throwing away all old software in a company to create a complete new information system as information infrastructure. Founding a new city can be done only where no one settled before or after a catastrophic event like an earthquake.

Following these points, one sees that an information infrastructure of a company which evolved through several years should be seen as a city grown for decades or even centuries. Of course, the growing of a city results in the need for modernizing the infrastructure and adding new city quarters and public buildings. However, changing the communication platform of a large information systems should be seen as similar to building a new underground transportation network in an old city rather than an algorithmic experiment.

The city metaphor fits well for another aspect: cities can be organized in various ways, and reorganizing a grown city like Calcutta is task different from modifying a planned city like Brasilia. The same can be observed for information systems which already have a history of some decades.

3 Adaptation in Database Applications

The need for software adaptation can be recognized in many typical database application scenarios. Database objects, for example in a bank application or production documents, may have a very long life-span. Some information has to be stored and manipulated for decades.

3.1 Example of Adaptive Applications

For example, let us consider the typical bank application where we have to store and manipulate information about accounts, customers, withdrawals etc. As an example, we consider **Account** objects with an **amount** attribute and typical events like **withdraw** and **deposit**.

Together with such information system objects we usually have a fixed part of manipulation functions, which is ideally stable for the life-span of the object (basic routines for manipulating attribute values, **withdraw** and **deposit** for our bank accounts). We call this part the fixed or *rigid* part of the object behavior. The rigid part is typically ‘hard-coded’ in database applications and realized by optimized code.

Other parts of a database application are subject to frequent change: constraints or business rules, exception activation and notification triggers. Rules for computing interests in a bank application or billing processes are examples of such changing parts. We call these parts *evolving*. These changes may result from changes in business processes and policies, but may even modify the behavior of single instances of object classes.

In typical database applications, the evolving part may contain the modification of the following application aspects:

- Constraints on the correctness of stored data may be adapted to new circumstances, for example the minimal age of people having a second credit card.
- Adapting language features for exception activation. Examples are alarming messages for certain withdraw patterns using a credit card.
- Similarly, some notification mechanisms may be adapted to situations.
- Also, the concrete rules for computing interests for saving accounts may change during the lifetime of account objects.

As these examples show, such adaptations may be due to several reasons:

- New insights or external changes in the modeled application enforce modifications of constraints and computation rules.
- Changes in the business processes and policies may create a need for new functionality.

As a last comment, one has to consider even changes for individual instances. In database applications, special constraints or exception triggers for single customers and their accounts are nothing special.

3.2 Evolution in SQL

Before we present our own proposal for handling adaptation on the design level, we will have a very brief look at how it is done nowadays on the implementation level (for a detail presentation of schema evolution features in SQL, we refer to [22]). A large part of the currently operational database applications are based on SQL database management systems. SQL supports a large variety of modifications during runtime of an application [8]:

- Most SQL functional units can be inserted, modified, and deleted at runtime, among them there are constructs like constraints, triggers, and stored procedures.
- The concept of stored procedures and triggers allows to describe functionality of an application in an explicit way allowing its manipulation at runtime PSM [15].
- SQL even allows the activation and de-activation at runtime during a transaction instance.

It should be noted that other database management system architectures support even more advanced features for adaptation. Object databases having a meta object layer support a more flexible (and maybe more dangerous) adaptation mechanism [17,4].

3.3 Our Approach: Explicit Handling of Evolution

After analyzing the requirements and current implementation techniques, we develop a method to prepare information systems for (restricted) evolution already at design time. This framework is based on the following concepts:

- During design, a *separation* of the rigid and the evolving part of application objects has to be performed.
- A base specification fixes the signature of application objects as well as basic functions.
- The evolution level manipulates (executable) specification fragments the vocabulary of which is identical to the vocabulary of the base level.
- All critical functions should be part of the rigid base level. These fixed functions are safe with respect to undesired modifications during evolution and their properties can be formally verified.

This separation can be found on all levels of system development as well as in the formal models for evolving objects:

- On the logical level we use an extended temporal logic allowing the explicit storing and manipulation of base level temporal formulae. The underlying signature is divided into a base and a mutation level. This separation is reflected in the semantic models, too.
- On the specification level we separate the rigid part from the evolution part using special keywords. This avoids confusing the levels during design. The flavor of the specification constructs, however, is the same for both levels.
- Even if the design is somehow independent of the later implementation techniques, we propose a similar separation there, too. The evolving part can be implemented using SQL features like triggers and stored procedures. During life-time of a running system, the maintenance and adaptation to new requirements will most of the times manipulate the evolving part!

4 Objects as Building Blocks

For conceptually describing information systems there is a large number of approaches. For instance, OMT [18] and UML [3] are well-known and frequently used in practice. These object-oriented modeling approaches provide a collection of mainly graphical description techniques (object diagrams, StateCharts, message sequence diagrams, etc). Unfortunately, their formal semantics is not completely clear, or, at least, there is no common understanding of their semantics. Other, more theoretical approaches provide a clean formal semantics based on well understood theoretical concepts. For example, TROLL [13] and Albert [9] belong to this class of conceptual modeling approaches.

All these approaches mentioned so far have some commonalities. First of all, they all consider objects as basic building blocks for information systems. In this way objects can be regarded as software units, or they correspond to software units. Another property of all these conceptual modeling approaches is that the behavior of the objects (and, thereby, of the entire system) is fixed at specification time. Usually, specification takes place before the system is implemented in order to be able to check the implementation against a conceptual specification. Not only the behavior of the single objects is fixed, but also the communication structure within the system. For instance, each object is given some communication channels to other objects for exchanging messages. However, these channels are also fixed at specification time. As we motivated in the introduction and as we will see in more detail in the next section, this is not adequate for information systems running for years or decades.

In this section we are going to introduce basic concepts for conceptual describing objects in information systems. Due to restrictions and lack of clarity concerning some modeling concepts of popular approaches like OMT and UML, we here use the formal specification language TROLL. In particular, if we want to have correctness as a major quality criterion, formal specification approaches are advantageous.

Object Specification

Due to the fact that there is a large number of concepts for object modeling and specification, we here focus on a number of essential concepts being available in most object-oriented modeling languages. First of all, objects in information systems have states. For that they have attributes the values of which may be changed in the course of time. Objects of the same type (having the same properties) are usually grouped into a class. Objects are then members of classes, the extension of a class consists of a collection of objects which may change as time evolves. Between classes we can have different kinds of relationships. Special kinds of relationships are specialization and generalization. Objects of the specialized class (the sub-class) inherit all properties of the super-class. The extension of a sub-class always is a subset of the extension of its super-class. Another kind of relationship is aggregation by which several objects of possibly different types become part of an aggregate object. These basic modeling

concepts allow to structure an information system into objects and classes. The permitted states of objects can be restricted by specifying constraints.

In addition to the structure of an information system we can also describe its intended behavior. For that, we can declare events which may occur and affect objects. Only the occurrence of an event may change the state of an object. The effect an event has to the state of an object is specified by valuation rules determining the change of attribute values. In addition, events may cause other events to occur in the same or other objects. This event calling can be seen as a communication primitive which enables message passing among objects. The occurrence of events can be restricted by enabling conditions stating in which state of an object a certain event may occur.

The semantics of such specifications can be defined in two levels. The first level deals with the single objects. Their semantics can for instance be given in a linear temporal logic. The second level is needed to compose the models of the single objects to a model of the whole system by respecting the structural restrictions (e.g., specialization between objects) and the communication between objects. For that several logical frameworks have been developed (e.g., [21]).

Object Specification Concepts

In order to provide a basic understanding of the behavior part of a TROLL specification, we briefly present some small examples:

- *Attribute valuation:*

The occurrence of an event in an object can change the values of the object's attributes. For that a rule can be specified describing the effect of the event on attributes, for instance:

decrease
changing Counter = Counter - 1;

- *Birth and death events:*

Events marked as **birth** or as **death** events play a special role in the life of an object. Only the occurrence of a birth event can start the life of an object in the information system, the occurrence of a death event stops the life of that object. There may be only one occurrence of a birth event (as well as only one occurrence of a death event) for each object. Before the birth event and after the death event no other event may occur.

- *Event permission:*

The occurrence of events may be restricted to certain states of an object. An enabling condition describes those states of an object in which the event may occur. If the condition is not fulfilled in the current state, the event may not occur:

decrease
enabled Counter > 0 ;

– *Event calling:*

The basic mechanism for communication in TROLL is event calling enforcing several events to occur simultaneously. A calling rule describes that the occurrence of an event causes the occurrence of another event:

`withdraw(m) calling Bank.increaseBookingCounter;`

Here, the occurrence of a **withdraw** event in an account object in a bank scenario calls the occurrence of the **increaseBookingCounter** event in the **Bank** object (which could be given by a reference within the account object). A special property of this calling mechanism is that the **withdraw** event is not permitted to occur if the **increaseBookingCounter** event is not enabled.

Example

The following example introduces a part of a scenario in which documents are managed in an information system.

object class Documents

identification DocId: (DocNo)

template

attributes

DocNo: int,

DocType: {offer, pre_contract, contract},

Valid: date,

Content: text;

events

birth create(DocNo:int,Content:text),

revise(NewContent:text),

prepare_contract,

fix_contract,

death resolve;

:



Fig. 1. Signature Specification in TROLL

Signature Specification in TROLL. Fig. 1 shows the signature part of a TROLL specification for **Documents**. In our example, documents can uniquely be identified by a **DocId** which is specified to be given by the attribute **DocNo**. Further attributes declared in the **attributes** section are **DocType**, **Valid**, and **Content**. There are three possible types for documents: **offer**, **pre_contract**, and **contract**. The attribute **Valid** specifies up to which day an offer document is valid. **Content** is the attribute containing the textual content of the

document. Beside these attributes and their data types, the signature part of on object (class) specification also contains the declaration of **events** (sometimes also called actions). The occurrence of an event can change the values of attributes and can cause the occurrence of other events in other objects (via communication). In the example there is a birth event **create**. Its occurrence creates a new objects and sets the initial state of that object. The **revise** event allows to change the contents of the document. The events **prepare_contract** and **fix_contract** are intended to change the type of the document. The occurrence of the **resolve** event deletes the object.

object class Documents

```

: // as specified in Figure 1
rigid axioms
  create(D,C)
    changing   DocType = offer,
                DocNo   = D,
                Content  = C,
                Valid    = now + 30
    calling    DocManager.addDocToOffers(self);
  revise(C)
    enabled    DocType = offer and Valid  $\geq$  now,
    changing   Valid    = now + 30,
                Content  = C;
  prepare_contract
    enabled    DocType = offer and Valid  $\geq$  now,
    changing   DocType = pre_contract,
                Valid    = now + 10,
                Content  = C;
:

```

Fig. 2. Behavior Specification in TROLL

Behavior Specification in TROLL. Fig. 2 shows the second part of the document class specification. In this part the behavior of document objects is fixed. For each event declared before in the signature part its effect on attributes (in the **changing** part), its enabling condition (in the **enabled** part, if a condition is given), and its communication effects (in the **calling** part) is specified.

Temporal Logic Interpretation

For such specifications of information systems different formal frameworks can be used for defining a semantics. In particular, linear temporal logic [10,14] can

be used as basic semantical level for TROLL [13], i.e. as semantics for the single objects. In order to define a semantics for a whole system of interacting objects the local models for objects have to be composed adequately respecting the communication between objects in order to build a model for the whole system. A corresponding language is for instance the *Object Specification Logic* OSL [21] which provides a framework for object systems where each single object is described by means of standard linear temporal logic.

In order to give an impression of using linear temporal logic as semantics for objects, we briefly present a few examples. The temporal logic operators **always** and **next** describe that the property which directly follows these operators holds in every future state or holds in the next state, respectively. The operator **occurs** is used to state the an event occurs in a state.

- The effect of an occurrence of the event **revise** on the attribute **Content** is described by the following temporal logic formula:

$$\forall C(\mathbf{always}(\mathbf{occurs}(\mathbf{revise}(C)) \Rightarrow \mathbf{next}(\mathbf{Content} = C)))$$

This formula says that the property, that if the event **revise** occurs with the actual parameter C in a state then in the next state the attribute **Content** has the value of C always, always holds (i.e., holds in every future state starting from the initial state).

- The permission condition for the event **prepare_contract** can be formulated in the following way:

$$\mathbf{always}(\mathbf{occurs}(\mathbf{prepare_contract}) \Rightarrow (\mathbf{DocType} = \mathbf{offer} \wedge \mathbf{Valid} \geq \mathbf{now}))$$

This formula states that it always holds that the occurrence of the event **prepare_contract** in a state requires that the attribute **DocType** has the value **offer** and the attribute **Valid** has a value greater or equal than the current date.

- The event calling specified for the event **create** can be formulated as follows:

$$\mathbf{always}(\mathbf{occurs}(\mathbf{create}(D,C)) \Rightarrow \mathbf{DocManager.occurs}(\mathbf{addDocToOffers}(\mathbf{self})))$$

Please note that this formula fixes the semantics of event calling such that the calling event **create** and the called event **addDocToOffers** in another object with name **DocManager** occur synchronously, i.e., the called event is forced to occur in the same state.

- A temporal integrity constraint like “The length of the document text must never decrease” could be expressed in linear temporal logic as follows:

$$\forall x(\mathbf{always}(\mathbf{length}(\mathbf{Content}) = x \Rightarrow \mathbf{next}(\mathbf{length}(\mathbf{Content}) \geq x)))$$

In our example (Fig. 1 and 2) we do not have such a temporal integrity constraint, but the language TROLL provides the means to specify such a constraints in an additional part of an object (class) specification.

[13] gives a more detailed presentation of linear temporal logic as a semantic basis for the object specification language TROLL.

5 Towards Evolving Objects

As we have seen in the previous section, current object specification technology allows the declarative specification of the universe of discourse. For that, it provides means to capture structural as well as behavioral aspects of information systems. However, the dynamic behavior is totally fixed at specification time. Thus, changes in the real-world cannot be represented adequately by these approaches. In particular, objects in information systems can have a very long life-span. For example, objects representing contract documents or accounts may resist for decades. During such a long period, there may occur a lot of changes in the environment which often make it necessary to adapt the object behavior to the new requirements. In case of contract documents, for example, we have to consider new laws which may have effects on the resolution of contracts. For instance, a contract can only be resolved under special circumstances. Since such kinds of changes cannot always be known in advance, we need a flexible mechanism to capture dynamically changeable behavior.

What we need – to represent the real-world and its evolution as precisely as possible in an information system – are *evolving* objects! That are objects which do not only change their object states but also may change their possible object state transitions (behavior).¹ Evolving objects are able to deal with evolving requirements; their behavior is dynamically changeable. In order to overcome the limitations described in the previous section, we have developed a new specification framework which supports the definition of evolving behavior. The main idea behind this extended framework is to consider object states as theories rather than simple value mappings. Consequently, in this context object evolution corresponds to a theory revision.

Evolving Behavior Specification

In our extended specification framework [20,23,6,7], we distinguish between *rigid* axioms and *evolving* axioms. The rigid axioms represent the part of the behavior specification which is fixed and must not be changed during the evolution of the objects. These are “stable” axioms such as assigning a value to an attribute or creating an object. Evolving axioms, on the other hand, represent the evolving behavior part. By adding and removing arbitrary axioms during runtime, we may dynamically change the behavior of an object.

There are different ways to model evolving behavior. In [23] we have presented three ways to deal with this problem at the language level. In the following, we will use a special attribute, called *axiom attribute*, to store the currently valid set of evolving axioms. Furthermore, we have introduced special events, called *mutators*, which mutate the object specification by changing the axiom attribute. That is, a mutator changes the behavioral description of the object

¹ In general, it should also be possible to extend the state space of an objects dynamically in order to be able to capture the evolution of objects more precisely. But this issue is beyond the scope of this paper.

(at the meta level). The parameter of a mutator is of a special type, called *spec*, which represents axioms. This type must not be used in the rigid axioms part.

In Figure 3 we sketch the basic constructs of our extended specification language. The example specification extends the specification given in Figures 1 and 2. We define one axiom attribute **Rules** and the mutators **add_rule** and **remove_rule**. The effects of the mutators are described in the *dynamic specification* section.

Clearly, the dynamic specification part as depicted in Figure 3 represents a “classic” extension to capture changes of the behavior. However, there can be several axiom attributes and corresponding mutators.

```

object class Documents
  identification DocId: (DocNo)
  template
    attributes
      :
    events
      :
    rigid axioms
      :
    axiom attributes
      Rules initialized { }
    mutators
      add_rule(Rule:spec)
      remove_rule(Rule:spec)
    dynamic specification
      add_rule(Rule)
        changing Axioms = Axioms + { Rule }
      remove_rule(Rule)
        changing Axioms = Axioms - { Rule }
  end object class

```

Fig. 3. Extended Specification Capturing Evolving Behavior

Note that in our framework the same language constructs are used for manipulating the base level as well as the meta level. For instance, the occurrence of mutator events can be restricted in the same way as usual events by defining enabling conditions.

Figure 4 depicts one possible life cycle of an evolving object (which corresponds to a document). As usual objects, evolving objects are created with a birth event, which is named **create** in this case. Thereafter, the state and/or behavior of this object may be changed in several ways. For instance, by calling

the event **revise** we may change the text of this document. By calling the mutator event **add_rule** we may add new rules to the document. And finally, we may destroy this document object by calling the death event **resolve**.

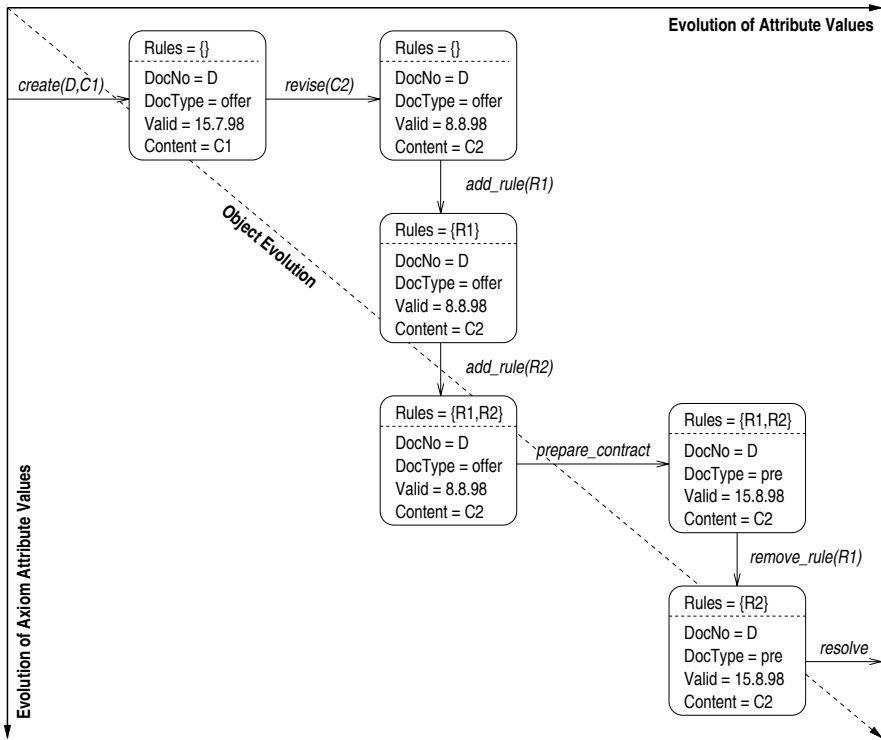


Fig. 4. Effects of Events and Mutators

Changing Dynamically the Behavior Specification

The behavior of a document object may be changed, for instance, by adding evolving axioms to or removing evolving axioms from the axiom attribute **Rules**. Adding new axioms to the specification means to further restrict the behavior of an evolving object. Analogously, removing axioms from the specification means to allow more possible behavior patterns. In the following we present some parameters for the mutators **add_rule** and **remove_rule**:

- Suppose, due to new requirements, it is necessary to store the date when a contract is fixed. This new behavior can be added to the current specification by using the mutator **add_rule** with the following parameter:

fix_contract

changing Valid = now

- A new business rule such as “*The supervisor object ‘DocManager’ must be informed in case of revisions*” is introduced by the following specification:

revise(C)

calling DocManager.inform_me(DocNo, C)

- We can also use a mutator to restrict the enabling of events. For instance, to keep contract document objects alive forever we can use the mutator **add_rule** with the following parameter:

resolve

enabled DocType \neq contract

- At runtime we can also add new integrity constraints restricting the allowed states of an evolving object. For example, the following axiom constrains a document to be an offer or a pre-contract if the validity date is set to a date in the future:

Valid > now \rightarrow

(DocType = offer **or** DocType = pre_contract)

As we have seen above, the extensions provide us a powerful specification framework. The question is now how to use this framework effectively. With the definition of the basic attributes and events we form the “shape” of an evolving object. Since the rigid axioms cannot be changed during runtime, we have to be very careful which part of the object behavior is specified as rigid axioms. On the other hand, if we specify the whole behavior as evolving axioms, we have the problem that “everything is possible”. In this case, we cannot prove any interesting property of evolving objects. Therefore, a deep analysis is required to find the part of behavior which is affected by the evolution of the object.



Corresponding Logic

For interpreting evolving specifications we have to go beyond first-order logic. In [6,5] we presented an extension of linear temporal logic called *Dynamic Object Specification Logic dyOSL*. *dyOSL* is based on the *Object Specification Logic (OSL)* presented in [21]. A full description of this logic is outside the scope of this paper. Therefore, we will only sketch the basic concepts of the formalization.

The logic OSL defines a linear temporal logic framework for object descriptions. Therefore, the semantics of an object description is a local object theory of a temporal logic. Both composition of objects and (monotonic) inheritance are described as operations to combine object theories. The logic *dyOSL* follows this framework and gives a local object theory for single evolving objects.

As in the specification language, we have a separation of base and meta level. Besides event and attribute symbols the signature for the logic contains the meta counterparts *MUT* for mutation event symbols and *MAT* for mutation

attribute symbols. The logic axioms are doubled for both layers following the spirit of OSL.

For appropriate models, we define a two-level interpretation structure where the base level is a usual temporal logic interpretation. The semantics of the mutations is defined analogously on the meta level. For combining these two levels, we introduce a special meta attribute Ax containing the current specification texts which are interpreted on the base level. In other words, the current value of Ax must be satisfied by the base level in the (relative) future of the object instance. □

The complete formalization of this logic can be found in [5] where also some remarks on proof techniques for such logics are given.

This solution has some consequences for dealing with evolving axioms containing the **always**-operator. If we add an axiom of the form **always**(ϕ) to Ax , this axiom will influence the complete future of the object. If we remove such an axiom afterwards, this will not effect the specification. Usually, one should only add state formulae to Ax in order to avoid such undesired effects.

Behavioral Changes to Single Objects or Whole Classes?

As far as we presented our proposal, we implicitly assumed that all changes by occurrences of mutator events refer to single objects. Since this might be surprising we here briefly discuss the question of granularity of such changes.

In our example, we introduced (the specification of) mutator events to the specification of a class **Documents**. Analogously to “normal” events, mutator events are specified as events occurring for single objects. In the same style as a **revise** event occurs for a single object — because it changes the state of a single object out of the class **Documents** — a mutator event **add_rule** occurs to a single objects. The language TROLL which we took as a basis for presenting our ideas does (on that level) not allow to specify events occurring to a whole class. Following the style of TROLL the way we introduced mutator events is the same in which events are specified.


Often the change of behavior, e.g. by adding a new axiom, should affect all objects of that class. One way to reflect this intention is to add a further specification means to the specification language for introducing events on the class level. Another way could be to specify, for example, a special object representing the whole class. For this object we then can declare mutator events as needed. Such events are sometimes called class events or class methods. For each such mutator event we must take care that by means of event calling the corresponding mutator event of each single object occurs simultaneously. In this way we can easily simulate the concept of (mutator) events on class level.

For using such a specification language in practice we think that both concepts, mutator events on object level as well as on class level, are needed. The concept of mutator events on class level allows an easy and uniform treatment of all objects of a class at the same time, whereas the concept of mutator events on object level enables us to deal, for example, with exceptions. Exceptions are used to model the fact that not all objects of a class must necessarily exhibit the same behavior.

6 Conclusions and Future Work

Information system design differs from program construction as much as house building differs from maintaining a large city. Information systems are long-living, evolving software constructs that integrate several partially autonomous subsystems. In this paper, we dealt with the evolution aspect of such systems and showed how evolution can be handled during a formalized modeling phase. In particular, we presented an extension of an object specification framework to capture evolution aspects as part of the conceptual description. We pointed out that building large information systems is a continuous (infinite?) process with changing requirements. In this sense, evolution of information systems can be seen as continuous adaptation to new requirements.

In order to get a grasp of the problem of behavior evolution, we argued for a separation of concerns. The language TROLL which is based on object orientation and temporal logic is extended to describe evolving objects without inventing new formalisms. The behavior specification is divided into a rigid part (which is optimized and hard-coded) and an evolving part (which can be changed at runtime). We introduced the concept of evolving objects. The core functionality which always has to be guaranteed is specified in the rigid part of such an object. The behavior which may be changed during the life-span of an evolving object is described by adding axioms to and removing axioms from a specification.

Currently, we are investigating different extensions of temporal logic for reasoning about such evolving specifications. These extended logics differ in the aspect how to interpret the state-dependent specification fragments. *dyOSL* [5] represents one possibility by interpreting them at runtime. On the opposite side, a compilation to a temporal logic without reflection but explicit mutation states seems possible as well. 

However, there remains a lot of open questions. One question concerns the separation of the rigid and the evolving parts in existing applications. Here, we have to focus on the consequences for re-engineering of information systems. Especially, several case studies are necessary to estimate how far behavior evolution can be modeled in advanced. A first case study not completely finished is described in [1].

Another very important issue is *signature evolution*. In particular, we have to analyze the effect of adding or dropping attributes/events to or from a specification, respectively. Does it make sense to add axiom attributes or mutators? Which kinds of behavior evolution are necessary and which ones should be forbidden?

References

1. S. Balko. *Adaptive Specification of Technical Information Systems*. In H. Balsters, B. de Brock, and S. Conrad, editors, *Proceedings of the Ninth International Workshop on Foundations of Models and Languages for Data and Objects: Database Schema Evolution*, LNCS, Springer-Verlag, Berlin, 2001 (this volume).
2. G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1991.

3. G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language (Version 1.0)*. Rational Software Corporation, Santa Clara, 1997.
4. S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules — The IDEA Methodology*. Addison-Wesley, Reading, MA, 1997.
5. S. Conrad, J. Ramos, G. Saake, and C. Sernadas. Evolving Logical Specification in Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 7, pages 199–228, Kluwer Academic Publishers, Boston, 1998.
6. S. Conrad and G. Saake. Extending Temporal Logic for Capturing Evolving Behaviour. In Z.W. Raś and A. Skowron, editors, *Foundations of Intelligent Systems, Proc. of the 10th Int. Symposium, ISMIS'97, Charlotte, North Carolina, USA, October 1997*, Lecture Notes in Artificial Intelligence, Vol. 1325, pages 60–71. Springer-Verlag, Berlin, 1997.
7. S. Conrad, G. Saake, and C. Türker. Towards an Agent-Oriented Framework for Specification of Information Systems. In J.-J. Ch. Meyer and P.-Y. Schobbens, editors, *Formal Models of Agents — ESPRIT Project ModelAge Final Report, Selected Papers*, Lecture Notes in Artificial Intelligence, Vol. 1760, pages 57–73. Springer-Verlag, Berlin, 1999.
8. C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, Reading, MA, 4 edition, 1997.
9. E. Dubois, P. Du Bois, and M. Petit. O-O Requirements Analysis: An Agent Perspective. In O. Nierstrasz, editor, *ECOOP'93 — Object-Oriented Programming, Proc. 7th European Conf., Kaiserslautern, Germany, July 1993*, Lecture Notes in Computer Science, Vol. 707, pages 458–481. Springer-Verlag, Berlin, 1993.
10. E. A. Emerson. Temporal and Modal Logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 995–1072, Elsevier Science Publishers, North-Holland, Amsterdam, 1990.
11. M. Gogolla, S. Conrad, G. Denker, R. Herzig, N. Vlachantonis, and H.-D. Ehrich. TROLL light: The Language and Its Development Environment. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, Lecture Notes in Computer Science, Vol. 1009, pages 205–220. Springer-Verlag, Berlin, 1995.
12. R. Herzig, S. Conrad, and M. Gogolla. Compositional Description of Object Communities with TROLL light. In C. Chrisment, editor, *Proc. of the Basque Int. Workshop on Information Technology (BIWIT'94)*, pages 183–194. Cépaduès-Éditions, Toulouse, 1994.
13. R. Jungclauss, G. Saake, T. Hartmann, and C. Sernadas. TROLL — A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.
14. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification*. Springer-Verlag, New York, NJ, 1992.
15. J. Melton. *Understanding SQL's Stored Procedures — A Complete Guide to SQL/PSM*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.
16. H. Müller and H. Weber, editors. *Continuous Engineering for Industrial Scale Software Systems*, 1998.
17. N. W. Paton and O. Díaz. Metaclasses in Object-Oriented Databases. In R. A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction, Proc. of the IFIP WG 2.6 Working Conf., DS-4, Windermere, UK, July 1990*, pages 331–347. North-Holland, Amsterdam, 1991.
18. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

19. G. Saake and T. Hartmann. Modelling Information Systems as Object Societies. In K. von Luck and H. Marburger, editors, *Management and Processing of Complex Data Structures, Proc. of the 3rd Workshop on Information Systems and Artificial Intelligence, Hamburg, Germany, February/March 1994*, Lecture Notes in Computer Science, Vol. 777, pages 157–180. Springer-Verlag, Berlin, 1994.
20. G. Saake, A. Sernadas, and C. Sernadas. Evolving Object Specifications. In R. Wieringa and R. Feenstra, editors, *Information Systems — Correctness and Reusability. Selected Papers from the IS-CORE Workshop*, pages 84–99, World Scientific Publishing, Singapore, 1995.
21. A. Sernadas, C. Sernadas, and J. Costa. Object Specification Logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.
22. C. Türker. *Schema Evolution in SQL-99 and Commercial (Object-)Relational DBMS*. In H. Balsters, B. De Brock, and S. Conrad, editors, *Database schema Evolution and Meta-Modeling — 9th International Workshop on Foundations of Models and Languages for Data and Objects (FOMLADO/DEMM 2000)*, Dagstuhl, Germany, Lecture Notes in Computer Science Vol. 2065, pages 1–32 Springer-Verlag, 2001.
23. C. Türker, S. Conrad, and G. Saake. Dynamically Changing Behavior: An Agent-Oriented View to Modeling Intelligent Information Systems. In Z. W. Raś and M. Michalewicz, editors, *Foundations of Intelligent Systems, Proc. of the 9th Int. Symposium on Methodologies for Intelligent Systems, ISMIS'96, June 1996, Zakopane, Poland*, Lecture Notes in Artificial Intelligence, Vol. 1079, pages 572–581. Springer-Verlag, Berlin, 1996.
24. R. J. Wieringa. A Formalization of Objects Using Equational Dynamic Logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object Oriented Databases, Proc. of the 2nd Int. Conf., DOOD'91, Munich, Germany, December 1991*, Lecture Notes in Computer Science, Vol. 566, pages 431–452. Springer-Verlag, Berlin, 1991.

Extending the Object Query Language for Transparent Metadata Access

Hong Su, Kajal T. Claypool, and Elke A. Rundensteiner

Worcester Polytechnic Institute, 100 Institute Road,
Worcester, MA, USA

{suhong, kajal, rundenst}@cs.wpi.edu
<http://davis.wpi.edu/dsrg>

Abstract. Many applications in Object Databases (ODB), for example, schema management tools, CASE tools, database development tools and integration wrappers, need extensive queries over both application data as well as metadata. However queries over metadata via OQL, a de-facto standard for object query languages defined for the ODMG 2.0 Object Model, are tied to low-level implementation details of the underlying schema repository of the database system. Hence, they are neither portable nor easily usable, requiring the application developer to have detailed knowledge of the proprietary structure of the schema repository. In this paper, we propose an extension of OQL, called MetaOQL, to address this limitation. Our proposition of MetaOQL offers several benefits: (1) it is a natural extension of OQL in terms of both its syntax and semantics; (2) it removes the dependency of metadata queries on the particular schema repository, hence providing uniformity and portability of metadata queries across different ODBs; (3) it supports transparent navigation over the metadata thus offering ease of use; (4) unlike OQL, it hides metadata querying details from the users hence the queries can be simplified and more easy to read and understand. We have also investigated implementation strategies for MetaOQL. In particular, we propose a translation strategy from MetaOQL to OQL as a preferable solution compared to development of a special-purpose MetaOQL processor. The translation strategy offers the advantage that the MetaOQL queries can be retargeted to work on top of any existing ODB engine equipped with OQL with minimal effort. Furthermore, all OQL query optimization strategies can thus still be brought to bear in our extended system.

1 Introduction

Motivation: Metadata Access. Analogous to system tables in relational database systems, metadata in object database systems (ODB) provides the descriptive information about the database objects that defines the schema of a database. A large number of applications on ODBs, especially schema management tools, CASE tools, schema evolution tools and integration wrappers, need to have access to the metadata. For example, class and object browsers [Obj99a] often

need to access metadata in order to display pertinent information such as the class definitions, the inheritance hierarchy or the association between classes. The schema evolution tools [CJR98a] dealing with schema restructuring such as adding new attributes to a class, adding new classes into a class hierarchy, involve extensive metadata querying and manipulation.

Most ODB systems indeed do allow access to the metadata [Tec94, Inc93]. As per ODMG, the access to metadata is often provided via a high-level declarative interface such as a query language instead of just a procedural low-level application programming interface (API) [Obj93]. An object query language, by virtue of treating the schema information as objects, is in principal capable of powerful metadata querying and manipulation. OQL [Cea97] is such a powerful standard object query language based on ODMG 2.0 Object Model that combines high-level declarative programming features with the object-oriented paradigm.

Problems of Metadata Access via OQL. As per the ODMG 2.0 standard [Cea97], metadata is stored in an Object Definition Language (ODL) schema repository, which is accessible to tools and applications using the same operations that apply to user-defined types. However, the ODMG standard only defines the interface methods through which meta-information is manipulated, rather than defining the entire class structure and details of the internal implementation of the schema repository. Thus, today while most commercial vendors [Obj99b, Gem99] attempt to provide ODMG compliant ODB systems, the actual physical representation of the schema repository varies from vendor to vendor. Hence while OQL provides declarative access to the schema repository, the OQL queries on metadata are tied to the vendor-specific schema repository. This has several disadvantages. First, the metadata query is not portable due to its tight coupling to the vendor-specific schema repository. Second, a user cannot query the metadata without complete knowledge of the internals of the schema repository of the given ODB system. Thus, tools such as class or object browsers today need to be implemented and deployed for a particular database. Portability of these tools to other ODB systems requires extensive re-engineering of the tools, or more development from scratch.

These disadvantages are also exemplified in the SERF system (Schema evolution through a Extensible, Re-usable and Flexible framework) [CJR98b]. Schema evolution is a fundamental aspect of information and database systems [Rea00]. SERF is the first system that enables users to define new complex schema evolution transformations in a flexible and extensible manner. Most systems support schema evolution by providing a set of schema evolution primitives. However, we found most schema evolution can be broken down to a sequence of minimal set of *basic schema evolution primitives*. Hence SERF uses OQL to arbitrarily combine these basic schema evolution primitives, application object updates as well as metadata access to describe the desired transformations. OQL queries over the schema repository to gather system schema information for use in the transformations. However relying on OQL alone for the metadata access often detracts from the portability of the transformations.

Example Illustrating the Problem. This problem is illustrated in the example below. Figure 1 depicts a complex transformation *inline* defined as the replacement of a referenced type with its type definition. In this case, the class *Person* has an attribute *address* of the complex type *Address*. In the transformation, the *Address* type is inlined into the *Person* class, i.e., all the attributes defined in the *Address* type (the referenced type) are now added to the *Person* type resulting in a more complex *Person* class. Figure 2 is an SERF representation of the *inline* transformation using OQL. In the transformation, it calls two basic schema evolution primitives provided by the underlying ODB system, *add_attribute*¹ and *delete_attribute*².

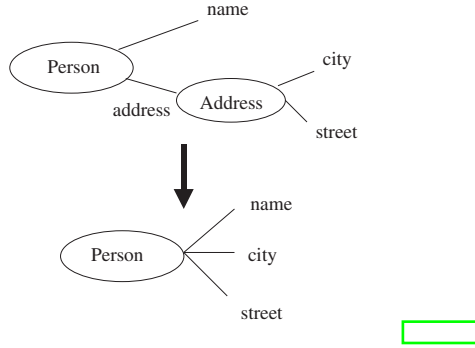


Fig. 1. Inline Transformation

The current SERF prototype is built using Object Design Inc.’s Persistent Storage Engine Pro 2.0 (PSE Pro) [O’B97] as the underlying ODB system. In the OQL statements in Figure 2, all words in bold font are tied to the implementation details of the structure of the underlying schema repository. Thus the codes of the transformations built on the proprietary schema repository of PSE Pro are not portable across different ODB systems due to the schema discrepancy. Suppose the application has another version built on another ODB and we also want to port this transformation built on PSE Pro ODB to the targeted ODB, we would have to modify all the system-dependent parts of the metadata queries to adjust to the specific physical representation of the targeted ODB. For example, first, the name of the class whose instances describe application class definition information (in our example, *MetaClass*) may be different in another ODB system (for instance, in ObjectStore [Obj93], *os_class_type*). Second, besides the name discrepancy which can be solved by simple string replacement,

¹ *add_attribute(String className, String attrName, String attrType)* will add a new attribute *attrName* of type *attrType* into the class *className*. If successful, it returns true, otherwise false.

² *delete_attribute(String className, String attrName)* will delete an attribute *attrName* from the class *className*. If successful, it returns true, otherwise false.

```

// Retrieve local attributes of class cName
define localAttrs(cName) as
  element(select c.localAttrList
    from MetaClass c
    where c.className = cName);

// Get type name of attribute aName defined in class cName
define refClass(cName, aName) as
  element(select attr.attrType.typeName
    from localAttrs(cName) attr
    where attr.attrName = aName);

//call schema evolution primitive add_attribute to add the
//attributes defined in address's type to class Person
for all attr in localAttrs(refClass(Person,address)) :
  add_attribute(Person, attr.attrName, attr.attrType);

//call schema evolution primitive delete_attribute to delete
//attribute address from class Person
delete_attribute (Person,address);

```

Fig. 2. Inline Transformation Written in OQL

the attribute *localAttrList* of the class *MetaClass* which describes the local attribute definitions may not be a public attribute in another ODB. In this case, we may need to identify and then make use of the method to retrieve the desired *localAttrList* information.

Another limitation arises when the metadata query is tightly coupled to the implementation details of the schema repository. While the developers of an application will have complete knowledge of the application schema since they designed it themselves, they may have little knowledge of the schema repository's specific structure. This would require them to resort to the system documentation of the ODB system to gain complete knowledge of the structure of the schema repository. Furthermore, metadata queries involving the retrieval details not only lead to bloated codes in many cases, but also are not too easy to understand, impacting the codes' readability and thus software maintainability which are very important criteria in the software industry.

Our Proposition. Hence there is a need for a general metadata access language to overcome these shortcomings. The language should be independent from the actual physical storage structure of the schema repository. We here propose a new language as an extension of OQL, called MetaOQL, that allows transparent system-independent navigation over the schema repository and hence provides uniform metadata access.

The advantages of MetaOQL are:

- MetaOQL is a natural extension to OQL, the de-facto standard of object query languages, in the sense that it is compatible with OQL syntax and semantics, hence it is easy to use for the users who are familiar with OQL;
- MetaOQL provides generic metadata queries portable across different ODMG compliant ODB systems;

- MetaOQL supports transparent navigation over the metadata thus providing better usability to users in that the users need not be aware of low-level details of the schema repository in order to get access to it;
- By hiding the internal metadata retrieval details from the users, metadata query statements can be drastically simplified and thus are more easy to understand, providing better readability to users;
- MetaOQL can be implemented in a non-intrusive manner on any current ODB system that supports OQL without imposing restrictions on ODBs.

To summarize, the contributions of this work include:

- We identify the limitation of the current object query language OQL with respect to metadata queries in terms of portability and usability;
- We propose a language that solves the problem of limited metadata query portability across different ODB systems while at the same time improving its usability;
- We demonstrate the advantages of MetaOQL via extensive examples;
- We propose a one-pass translation approach from MetaOQL into OQL which is non-intrusive and highly efficient to implement MetaOQL on existing OQL engines.

The following outlines the remainder of the paper. Section 2 presents the syntax of MetaOQL while Section 3 illustrates the features of MetaOQL via examples. Section 4 describes the implementation strategy of MetaOQL. Section 5 discusses related research. We conclude with a summary and a discussion of future work in Section 6.

2 MetaOQL Syntax Extension to OQL

2.1 OQL Reviewed

OQL is an object query language proposed for the ODMG Object Model [Cea97]. It is similar to SQL92 with object-oriented extensions like complex objects, object identity, path expressions, operation invocation etc. And OQL is an expression language. A query expression is built from typed operands composed recursively by operators. Our MetaOQL extension to OQL introduces new syntax for representing special set expressions. So below we will quickly review the related parts of the OQL syntax while full details can be found elsewhere [Cea97].

OQL provides high-level primitives to deal with the *collection* construct. Collection objects are composed of distinct elements of the same type. *Set* type is one of the collection types supported by the ODMG Object Model. A set object is an unordered collection of elements, with no duplicates allowed. Set construct supports some operations that other collection types do not support.

The *collection expressions* include *universal quantification*, *existential quantification*, *membership testing* and *aggregate operators*. The *binary set expressions* include *union*, *intersection*, *difference* and *inclusion*. Typical syntax related to collection and set expressions as used later in our examples is listed below:

1. Universal quantification: *for all x in e_1 : p*

If x is a variable name, e_1 denotes a *collection* and p is an predicate expression of type *boolean*, then “*for all x in e_1 : p* ” returns true if all the elements of collection e_1 satisfy p and false otherwise.

2. Member testing: *x in e_1*

If e_1 denotes a *collection*, x is an object or literal having the same type or a subtype as the elements of e_1 , then “ *x in e_1* ” returns true if element x belongs to collection e_1 and false otherwise.

3. Binary set expression: If e_1 and e_2 are *sets*, $\langle \text{op} \rangle$ is an operator from *union*, *except*, *intersect*, $e_1 \langle \text{op} \rangle e_2$ computes set theoretic operations, union, difference, and intersection on e_1 and e_2 .

4. Select from where clause: select f from x_1 in e_1 , x_2 in e_2 , ..., x_n in e_n [where p]

The e_i ’s have to be of type *collection*, p has to be of type *boolean*. x_i is a variable that ranges over the collection e_i . The result of the query will be a collection of t , where t is the type of the result of function f .

2.2 Syntax of MetaOQL

MetaOQL extends OQL in that it introduces new set expressions for representing and manipulating metadata information. The definitions of these *MetaOQL-specific set expressions* are given below. We get some inspiration for some of the expressions, i.e. expression 1 (a), 1 (b) from ShemaOQL [LSS96]. Figure 3 shows an example of the ODL definition for a schema that will be used as our running example.

```

class Person
{
    attribute string name;
    attribute Address address;
}

class Graduate: Student
{
    attribute short graduateYear;
    attribute string street;
}

Class Student: Person
{
    attribute short id;
}

class Address{
    attribute string city;
    attribute string street;
}

```

Fig. 3. ODL Definition of a Database Schema

Definition 1 MetaOQL-Specific Set Expression

A *MetaOQL-specific set expression* is one of the following expressions, where cl is a *class expression* defined in Definition 2.

(a) \rightarrow

\rightarrow denotes a set of names of all classes that are defined in a given scope.

Currently, many ODB systems [Obj93] only support one single schema, i.e., all classes defined in a database are within the same schema scope (name space).

(b) $cl->$

$cl->$ denotes a set of *full path attribute expressions* of the local attributes defined in the class cl . The type of *full path attribute expression* is a new type we introduce into the MetaOQL type system.

(c) $cl->^*$

$cl->^*$ denotes a set of *full path attribute expressions* of the attributes, local and inherited, of a given class cl .

(d) $cl+$

$cl+$ denotes a set of names of the immediate superclasses of a given class cl . For an object model that does not support multiple-inheritance, it returns a singleton set composed of one single element.

(e) $cl++$

$cl++$ denotes a set of names of all the superclasses of a given class cl .

(f) $cl-$

$cl-$ denotes a set of names of the immediate subclasses of a given class cl .

(g) $cl--$

$cl--$ denotes a set of names of all the subclasses of a given class cl .

Example 1. In the schema defined in Figure 3, we have:

$->$ denotes a set of class names: {"Person", "Student", "Graduate", "Address"}.

$Person->$ denotes a set of full path attribute expressions of all local attributes defined in class Person: {Person.name, Person.address}.

$Student->^*$ denotes the set of full path attribute expressions of all local and inherited attributes defined in class Person: {Student.name, Student.address, Student.id}.

$Graduate+$ denotes the set of all immediate superclass names: {"Student"}

$Graduate++$ denotes the set of all superclass names: {"Student", "Person"}.

$Person-$ denotes the set of the immediate subclass names: {"Student"}.

$Person--$ denotes the set of all subclass names: {"Student", "Graduate"}.

Definition 2 Class Expression

A *class expression* cl is defined by one of the following forms:

- a constant, i.e., a fixed class name.
- an expression of applying a *get-type operator* (defined in Definition 4) to an *attribute expression* (defined in Definition 3).

Definition 3 Attribute Expression

A *attribute expression* is defined by one of the following forms:

- a , where a denotes a variable that ranges over the metadata set defined in Definition 1 (b) or 1 (c), i.e., $cl->$ or $cl->^*$ for some cl .
- $C.A_1.A_2...A_n$, where C denotes a fixed class name, $A_1, A_2, ..., A_n$ denote fixed attribute names. The expression is a concatenation of a class name and a sequence of attribute names.

Definition 4 Get-Type Operator

$@e_1$, where $@$ is referred as *get-type* operator, e_1 is an *attribute expression*, returns attribute e_1 's type name.

Example 2. $@(\text{Person.address}) = \text{"Address"}$

Definition 5 Get-Name Operator

e_1 , where $|$ is referred as *get-name* operator, e_1 is an *attribute expression*, returns the attribute e_1 's name.

Example 3. If *attr* is a variable that ranges over Person- and the current value of *attr* over the iteration is *Person.address*, then $|attr| = \text{"address"}$.

OQL is a purely functional language that allows its operators to be freely composed, as long as the operands respect the type system. This is a consequence of the fact that the result of any query has a type that belongs to the ODMG type model and thus can be queried again. As a natural extension to OQL, the new set expressions introduced by MetaOQL are also typed. Besides the set expressions $cl->$ and $cl->^*$ defined in Definition 1 (b) and 1 (c), all the other set expressions are of type *set<String>*. *Class Expression* is of type *String* with a constraint that the string must be a class name defined in the schema. For the expressions $cl->$, $cl->^*$ and *attribute expression*, we introduce a new type *full path attribute expression type*. A expression of this type is different from the concept of a *path expression* in OQL. A *path expression* is a way to navigate from a complex object to other object instances using object references to reach the desired data. It models a navigation over object instances. A *full path attribute expression* is a navigation over the class hierarchy to reach a desired attribute. For example, we have a class **Person** with the ODL definition shown in Figure 3 and an instance **p** of class **Person**, **p.name** is a *path expression* while **Person.name** is a *full path attribute expression*. Hence $cl->$ and $cl->^*$ are of type *set<full path attribute expression type>*. Example 5 shows the free composition of operators.

Example 4. $(@(\text{Person.address}))->$ denotes a set: {Address.city, Address.street}

Explanation: $@(\text{Person.address})$ will return the type name of attribute *address* defined in class *Person* which is *Address*. Then $(@(\text{Person.address}))->$ will evaluate as Address- which will return the full path attribute expressions of all attributes defined in type *Address*, i.e., *Address.city* and *Address.street*.

3 Portability and Usability Improved by MetaOQL

In this section, we illustrate some of the benefits of using MetaOQL in the place of writing OQL queries against the metadata repository of the respective ODB systems. We base our case study on the SERF system [CJR98a,CR99b,CR99a]. In its first cut, the SERF framework allows the users to describe complex schema transformations using OQL. These transformations are principally as general and hence applicable to any tool that is based on OQL requiring metadata access.



However, ad-hoc transformations suffer from the fact that they specify the transformation for one particular application schema only. To improve the re-usability of the transformations hence avoiding to rewrite each such transformation from the scratch, the notion of SERF templates [CJR98b,RCL⁺99], has been introduced. A template, a generalized named transformation that includes input and output parameters, can be instantiated and then applied to different systems and different application schemas based on the provided input parameters. Besides the significant advantages of the re-usability brought by SERF templates, the aim for these templates is also to make them portable over all ODMG compliant ODB systems in the form of libraries. However, the encapsulation and generalization of the transformation templates requires extensive metadata access based on the input parameters such as the given class and attribute names. While depending on OQL alone, the transformation templates are tightly coupled to a specific schema repository of an ODB system. Hence for SERF, MetaOQL brings forth the major advantage of portability. SERF templates becomes portable and reusable across different ODMG compliant ODBs as libraries.

Figure 4 illustrates an instantiated *inline* template to perform the same transformation shown in Figure 2. The template *Inline(String className, String refAttrName)* is a generalized transformation that inlines the type of an attribute specified by the parameter *refAttrName* into the class specified by the parameter *className*. In the example shown in Figure 4, the parameter *className* is instantiated to “*Person*” and parameter *refAttrName* to “*address*”.

```

Inline("Person", "address")
{
    for all attr in (@(Person.address)) -> :
        add_attribute ("Person", |attr|, @attr);

        delete_attribute ("Person", "address");
}

```

Fig. 4. Inline Transformation from Figure 2 Written in MetaOQL

Figure 5 depicts two other complex schema evolution transformations called *merge_difference* and *merge_union*. In the *merge_difference* transformation, the structure of the new class *JournalPaper* is defined by the difference of the attributes of the two source classes *Author* and *Paper* while in *merge_union* transformation, *JournalPaper* is defined as the union of the attributes of *Author* and *Paper*.

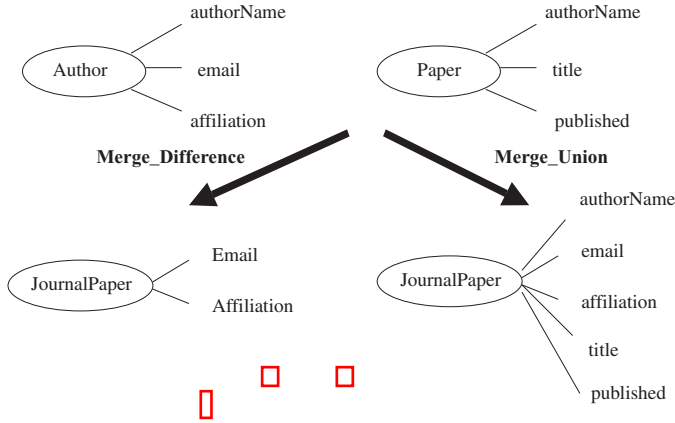


Fig. 5. Merge_Difference and Merge_Union Transformation

Figures 6 and 7 show the instantiated *merge_difference* transformation template³ written in OQL and MetaOQL respectively. Figures 8 and 9 show the instantiated *merge_union* transformation template⁴ written in OQL and MetaOQL respectively.

We can see that in the transformations written in OQL, named queries over the metadata are tightly coupled to the internal structure of schema repository and hence not so easy for the programmers to write and for the readers to understand. However in the transformation written in MetaOQL, all the named queries over metadata in OQL are now replaced by simple and clear MetaOQL-specific set expressions. From the comparison of the same transformation written in OQL and MetaOQL, it is obvious that the transformation can be drastically simplified if written in MetaOQL while at the same time being elegant and easy to read. And the uniform access to metadata also makes the transformation portable across different ODB systems.

4 The MetaOQL Translator

In the view of the popularity of the de-facto standard object query language OQL, we aim at an efficient yet non-intrusive implementation of MetaOQL. Our approach is thus to add a translator above the existing OQL engine instead of developing a new query processor. The translator will do a one-pass translation from MetaOQL statements to OQL statements. The later can then be processed

³ *merge_difference*(String className1, String className2, String newClassName) defines the transformation of merging difference attributes in class *className1* and class *className2* into new class *newClassName*.

⁴ *merge_union*(String className1, String className2, String newClassName) defines the transformation of merging union attributes in class *className1* and class *className2* into new class *newClassName*.

```

Merge_Difference("Author","Paper","JournalPaper")
{
  //Retrieve local attributes of class cName
  define localAttrs(cName) as
    element(select c.localAttrList
              from MetaClass c
              where c.className = cName);

  //Retrieve the difference attributes of class1 an class2
  define diffAttrs(class1,class2) as
    select attr1
    from localAttrs(class1) attr
    where not
      (attr1.attrName in
       select attr2.attrName
       from localAttrs(class2) attr2);

  //call schema evolution primitive to create a class JournalPaper
  create_class ("JournalPaper");

  //call schema evolution primitive to add difference attributes
  //of class Author and Paper into new class JournalPaper
  for all attr in unionAttrs("Author","Paper") :
    add_attribute(JournalPaper,attr.attrName,attr.attrType);
}

```

Fig. 6. Merge.Difference Transformation Written in OQL

by any of the existing OQL engines, exploiting their existing techniques for query optimization. The translator, which is a middleware, can be developed either by the ODB vendors or some other parties as long as the vendors provide the corresponding APIs we'll address later. In this way, we alleviate the additional requirement that are imposed on the ODB vendors. An additional advantage of adopting this translator approach is that not only the application written in the MetaOQL language but also the MetaOQL processor itself is portable to any OODB system.

4.1 Overview of the ODMG Schema Access Class Hierarchy

In the following, we expect the underlying ODB schema repository to be ODMG compliant. We briefly give an overview of the ODMG schema access class hierarchy as described in [Cea97].

- *d_Scope*

The *d_Scope* instances are used to form a hierarchy of meta-objects. A *d_Scope* instance contains a list of *d_Meta_Object* instances representing the elements defined in the schema. Operations to manage the list, e.g., finding a *d_Meta_Object* by its name, are provided.

- *d_Meta_Object*

Instances of *d_Meta_Object* are used to describe elements of the schema. In particular, we have:

```

Merge_Difference("Author","Paper","JournalPaper")
{
  //Retrieve the difference attributes of class1 and class2
  define diffAttrs(class1,class2) as
    select attr1
    from class1-> attr1
    where not
      (|attr1| in select |attr2| from class2-> attr2);

  create_class ("JournalPaper");

  for all attr in diffAttrs("Author","Paper") :
    add_attribute ("JournalPaper",|attr|,@attr);
}

```

Fig. 7. Merge_Difference Transformation Written in MetaOQL

- *d_Type*
d_Type is an abstract class for all type descriptions.
 - * *d_Class*
 A *d_Class* instance is used to describe an application-defined class. All persistent-capable classes are described by a *d_Class* instance.
 - * *d_Primitive_Type*
 A *d_Primitive_Type* represents all built-in types, e.g., short, float, boolean etc.
- *d_Attribute*
 A *d_Attribute* instance describes an attribute of an object or structure.

4.2 System Requirements

As we mentioned before, the responsibility of metadata retrieval is shifted from the users to the translator. The new expressions introduced in MetaOQL for referring to meta objects and their properties are effective shortcuts to retrieving metadata information. Users directly use these MetaOQL-specific expressions to represent sets of desired metadata without having to be concerned with how to retrieve such information from the system dictionary. The details of retrieval are left to the MetaOQL processor.

Since the translator itself is tied to the low-level implementation details of the underlying schema repository, one may expect that we have to develop a specific translator for each ODMG compliant ODB. This is obviously not an attractive idea. Here instead we put forth that we can develop a generic translator tool that can translate the MetaOQL metadata query statements into OQL statements specific to any schema repository. However for the uniform translator approach, we impose some system requirements on the ODMG compliant ODBs. There is a tradeoff between the uniformity of the translators built above different ODBs and the additional system requirements on the ODBs.

```

Merge_Union("Author","Paper","JournalPaper")
{
  //Retrieve local attributes of class cName
  define localAttrs(cName) as
    element(select c.localAttrList
              from MetaClass c
              where c.className = cName);

  //Retrieve the difference attributes of class1 and class2
  define unionAttrs(class1,class2) as
    localAttrs(class1) union localAttrs(class2);

  //call schema evolution primitive to create a class JournalPaper
  create_class ("JournalPaper");

  //call schema evolution primitive to add union attributes
  //of class Author and Paper into new class JournalPaper
  for all attr in unionAttrs("Author","Paper") :
    add_attribute("JournalPaper",attr.attrName,attr.attrType);
}

```

Fig. 8. Merge_Union Transformation Written in OQL

```

Merge_Union("Author","Paper","JournalPaper")
{
  //Retrieve union attributes of class1 and class2
  define unionAttrs(class1,class2) as
    class1-> union class2->

  create_class ("JournalPaper");

  for all attr in unionAttrs("Author","Paper") :
    add_attribute ("JournalPaper",|attr|,@attr);
}

```

Fig. 9. Merge_Union Transformation Written in MetaOQL

The ODMG 2.0 standard defines interfaces for accessing the schema of an ODMG database. The interfaces define an iterator protocol which supports several methods including retrieving the current element of the iterator, advancing the iterator to the next element, and so on. The class *MetaScope* defines an iterator that iterates over all instances of *MetaObject* defined in the schema. The class *MetaClass* defines an iterator that iterates over instances of *MetaAttribute* representing all local attributes defined in the class. The *MetaClass* also defines iterators that can iterate over the instances of *MetaClass* representing the subclasses and superclasses of the class itself represents.

Although we can retrieve all the meta-information via iterators, we instead expect the underlying ODBs to provide a set of schema access APIs that will make the translation process uniform. These schema access APIs can be implemented via the methods that should be provided by all ODMG compliant ODB

schema repositories to accord with the interface specifications. Since both the ODB vendors and any other programmers can implement these APIs easily as long as the underlying schema repository is ODMG compliant. Thus the uniform translator approach requires wrappers to be built by each ODB system. Hence for each ODB, there should be one such mapping table in which each MetaOQL-specific expression has an entry indicating the corresponding ODB API available to realize the desired functionality. Table 1 shows the mapping table we have built on the PSE Pro’s ODMG compliant schema repository, the system that we are targeting for implementing the MetaOQL processor. In addition, we also require system APIs that can retrieve certain common properties of the metadata objects. These methods are listed in Table 2.

Table 1. Schema Access APIs for PSE Pro

Expression	System API
->	MetaScope.getClasses
cl->	MetaClass.getLocalAttrs
cl->*	MetaClass.getAllAttrs
cl+	MetaClass.getSupers
cl++	MetaClass.getAllSupers
cl-	MetaClass.getSubs
cl - -	MetaClass.getAllSubs

Table 2. Methods to Retrieve Properties of Meta-Object

Desired Property	System API
className	MetaClass.getName
primitiveTypeName	MetaPrimitiveType.getName
attributeName	MetaAttribute.getAttrName
attributeType	MetaAttribute.getAttrType

For each metadata set expression, our MetaOQL wrapper for the PSE Pro schema repository provides a system API to retrieve the desired metadata. For example, the class *MetaScope* provides a method *getClasses* to retrieve the names of all classes defined in the schema scope. Another ODB may not provide APIs of exactly the same format as PSE Pro. Wrappers for ObjectStore [Obj93] for example provides a system API *os_database_schema.get_Classes* to realize the same functionality. Thus the implementation of our translator includes a graphic user interface (GUI) for the mapping table definition. The translator will translate MetaOQL metadata queries to OQL queries on the given proprietary schema repository using the APIs provided by the MetaOQL wrapper of the ODB system. And then the OQL statements will be processed by the existing OQL engine

to query and manipulate the metadata. The procedure of processing MetaOQL by the generic MetaOQL processor is shown in Figure 10.

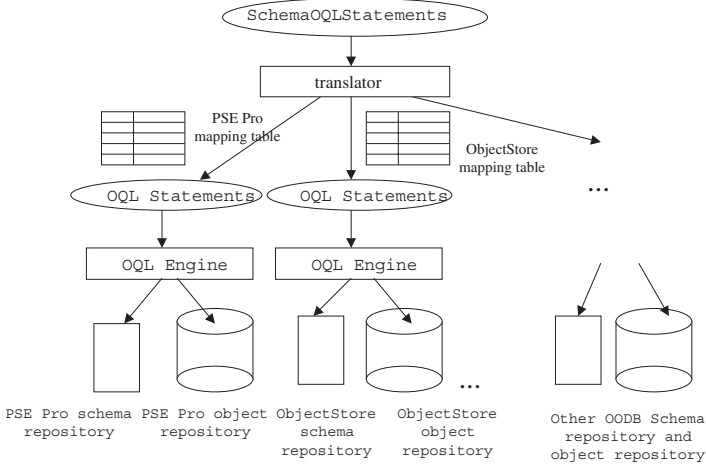


Fig. 10. One-Pass General MetaOQL Translator Approach

4.3 Translation Strategy

The translation process consists of two steps:

Step 1: Generate all named queries according to the mapping table and put all these system-specific named queries at the head of OQL statements;

Step 2: Scan the MetaOQL statements, replacing all MetaOQL-specific set expressions and operators with their corresponding defined named queries as further explained below.

It can easily be seen that our translation from MetaOQL to OQL is a one-pass process without any intermediate execution. In the following, we will discuss the generating the named queries basing on the mapping tables of Table 1 and 2 provided by the wrappers for PSE Pro schema repository.

Translation of MetaOQL-Specific Set Expression. For each MetaOQL-specific set expression, we define a named query. We use each name query to replace a MetaOQL-specific set expression during translation process. In OQL, a named query “*define [query] id(x_1, x_2, \dots, x_n) as $e(x_1, x_2, \dots, x_n)$* ” records the definition of the function with name *id* in the database schema. Once the definition has been made, each time when the OQL query engine evaluates a query and encounters such a function expression, if it cannot be directly evaluated or bound to a function or method, the OQL query engine replaces the query name *id* by the expression *e*. We use the schema access APIs exposed in the MetaOQL

wrapper of the specific ODB in these named queries to access the metadata. Due to the space limitation, we only list the named queries for the metadata set of class names, the full path attribute expression of local attributes (similar to the metadata set of full path attribute expression of local and inherited attributes) and the immediate superclass names (similar to the metadata set of all superclass names, immediate subclass names and all subclass names). All the bold words in the following named queries vary with different wrapper APIs.

1. \rightarrow metadata set of class names defined in the schema
2. $cl\rightarrow$ metadata set of full path attribute expressions of local attributes defined in a class

```
define getClasses(CName) as
  element(select c
            from MetaClasses 6 c
            where c.getClassName = CName).getLocalAttrs;
```

It should be mentioned here that the new type *full path attribute expression type* is introduced here. We mentioned before that set expressions $cl\rightarrow$ and $cl\rightarrow^*$ are of type *set<full path attribute expression type>*. For users who write in MetaOQL, it is a new type that we introduced into the MetaOQL type system. However, at the implementation level, it is in fact a system dictionary defined type *MetaAttribute*. The reason that we do not introduce it to the users as type *MetaAttribute* is that we do not want to expose the concept of *meta-object* to the users in order to hide the low-level implementation details of the schema repository. The users do not need to be aware of the internal details of meta-objects at all. MetaOQL supplies an easy mechanism to refer to *MetaAttribute* objects by *full path attribute expression*. The new type is more intuitive to the users and easier to understand than the concept of *meta-object*.

3. $cl+$ immediate superclass name set

```
define immediatesuperClass(CName)
  element(select c
            from MetaClasses c
            where c.getClassName = CName).getSupers;
```

Translation of *get-type* Operator. For translating *get-type* operator, we define two named queries for the *get-type* operator. Both named queries return the type name of a given attribute. The first query definition *getTypeFrName* takes two parameters *CName* and *AttrName*. The parameter *CName* is a class name and the parameter *AttrName* is the name of an attribute defined in the class specified by the parameter *CName*.

```
define getTypeFrName(CName,AttrName) as
  element(select attr.getAttrType.getName
            from getLocalAttrs(CName) attr
            where attr.getAttrName = AttrName);
```

⁵ *MetaScopes* is the extent of class *MetaScope*. This is a naming convention in ODB.

⁶ *MetaClasses* is the extent of class *MetaClass*.

The second query definition *getTypeFrMeta* only takes one parameter *AttrMeta*. The parameter *AttrMeta* will be bound to a full path attribute expression.

define getTypeFrMeta(AttrMeta) as MetaAttr.getAttrType.getName;

Then, when the translation processor encounters a *get-type* operator,

- if the operand is of the form *a* where *a* is a variable that ranges over the metadata set defined in Definition 1 (b) or 1 (c), replace *@a* with *getTypeFrMeta(a)*.
- if the operand is of the form *C.A₁.A₂...A_n* where *C* is a fixed class name and *A₁, A₂, ..., A_n* are fixed attribute names, replace *@(C.A₁.A₂...A_n)* with the following expression:

```
getTypeFrName(getTypeFrName(...
               getTypeFrName(getTypeFrName(C,A1),A2)...),An).
```

Example 5. *@(Person.address.city)* is translated into the following expression:
`getTypeFrName(getTypeFrName('Person', 'address'), 'city');`
`getTypeFrName('Person', 'address')` will return the type name of the attribute *address* defined in class *Person*, i.e., "Address", and then `getFrName('Address', 'city')` will return the type name of attribute *city* defined in class *Address*, i.e., "String".

Translation of *get-name* Operator. For translating the *get-name* operation, we define the following named query *getAttrName* with a parameter *AttrMeta*.

define getAttrName(MetaAttr) as AttrMeta.getAttrName;

Then the processor translates *|attr|* where *attr* is an variable of type *MetaAttribute* into *getAttrName(attr)*.

5 Related Work

XSQL [KKS92] is a language that is capable of querying and restructuring ODBs. It has an SQL like syntax and can express sophisticated queries in a concise way. This is achieved via extended path expressions which may have variables that range over classes, attributes, and methods. Unlike OQL, it is possible to query data without complete knowledge of the schema. However, the complex nature of XSQL raises concerns about effective and efficient implementation, a concern not addressed in their work.

Also in the relational world, several papers have appeared in the literature (e.g. [KLK88,CKW89,KLK91,LSS93,KLJ95]) that address the meta-data dependency problem. The solutions proposed in [KLK88,CKW89,KLK91] augment the query language with mechanisms that allow it to query both meta-data and ordinary data. These solutions are embedded in very powerful object-oriented query languages. Following the work [LS93], schemaSQL [LSS96,LSS99] is a recently proposed extension to SQL designed for multi-database interoperability. SchemaSQL retains the flavor of SQL while supporting the manipulation of data and metadata. SchemaSQL permits four additional types of variables in the *from* clause: db-name, rel-name, attr-name and domain variables besides

the tuple variables SQL already supports. The db-name variable ranges over a set of database names in a multi-database federation. The rel-name variable ranges over a set of relation names in a database. The att-name variable ranges over a set of names of attributes in a relation. The domain variable ranges over a set of values appearing in a column in a relation.

MetaOQL differs from SchemaOQL in the following aspects:

- SchemaOQL only queries the metadata of schema labels, i.e., relational labels and field labels. It does not query other meta-information for example a field's type. Only querying the those counterpart in ODB is not sufficient for supporting schema evolution. MetaOQL supports access to richer meta-information that are needed due to the object-oriented nature of the underlying data model such as an attribute's type besides its name;
- SchemaSQL focuses on the issue of interoperability between heterogeneous databases while MetaOQL focuses on improving portability and usability of the object query language. Hence the implementation of SchemaOQL focuses on conversion between metadata and data while that of MetaOQL focuses on translating general meta-query to system-specific query.

6 Conclusions and Future Work

Summary. In this paper we proposed a new language, MetaOQL, that addresses the limitations of portability and usability of metadata querying in ODBs. We introduce new set expressions for meta-information. These expressions are more than syntactic sugaring. Our proposal of MetaOQL removes the tight coupling between metadata retrieval and the actual physical storage structure of the schema repository. Metadata queries are no longer tied to any proprietary schema repository. And the users do not need to be aware of low-level implementation details of the schema repository. They can write more concise, simple and transparent queries on metadata. Moreover, MetaOQL is a natural extension to OQL and can be implemented in a non-intrusive manner by adding a preprocessor (translator) above any existing OQL engine. Our proposed translator approach is:

- Efficient: The translation can be done as a one-pass process in a preprocessing phase;
- Non-intrusive: No extension of standard OQL processor is needed;
- Generic: Simply by providing a mapping table, the translator and hence MetaOQL would work for any ODMG compliant ODB without requiring any software development.

With the development of ODBs, more and more schema features are adopted by the ODMG standard. For instance, key as an integrity constraint has been supported in ODMG 2.0. Before that there is no ODB vendors providing schema evolution primitives supporting the transformations such as adding and dropping keys (users have to hardcode to ensure the integrity constraint while performing the schema evolution). Therefore, with new in-coming features embraced by the standard, we need to explore what other operators we may need to support the corresponding schema evolution.

References

- [Cea97] R. G. G. Cattell and D. Barry et al. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.
- [CJR98a] K. T. Claypool, J. Jin, and E. A. Rundensteiner. OQL_SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In *Centre for Advanced Studies Conference*, pages 108–122, November 1998.
- [CJR98b] K. T. Claypool, J. Jin, and E. A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. Technical Report WPI-CS-TR-98-9, Worcester Polytechnic Institute, May 1998.
- [CKW89] W. Chen, M. Kifer, and D.S. Warren. Hilog as a platform for database languages. In *DBPL*, pages 37–44, 1989.
- [CR99a] K. T. Claypool and E. A. Rundensteiner. Flexible Database Transformations: The SERF Approach. In *IEEE Data Engineering Bulletin 22(1)*, pages 19–24, 1999.
- [CR99b] K. T. Claypool and E. A. Rundensteiner. SERF: An ODMG Based Extensible Schema Transformation System. Technical Report WPI-CS-TR-99-21, Worcester Polytechnic Institute, July 1999.
- [Gem99] GemStone. <http://www.gemstone.com/>, 1999.
- [Inc93] Itasca Systems Inc. Itasca Systems Technical Report. Technical Report TM-92-001, OODBMS Feature Checklist. Rev 1.1, Itasca Systems, Inc., December 1993.
- [KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. *SIGMOD*, pages 393–402, 1992.
- [KLJ95] M. Kifer, G. Lausen, and Wu. J. Logical foundations of object-oriented and frame-based languages. In *JACM*, pages 741–843, 1995.
- [KLK88] R. Krishnamurthy, W. Litwin, and W. Kent. Towards a real horn clause language. In *VLDB*, 1988.
- [KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Languages features for interoperability of databases with schematic discrepancies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 40–49, 1991.
- [LS93] K. W. Liu and D. Spooner. Object-oriented database views for supporting multidisciplinary concurrent engineering. In *IEEE Computer Software and Applications Conference*, pages 19–25, November 1993.
- [LSS93] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database system. In *DOOD*, pages 81–100, 1993.
- [LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *VLDB*, pages 239–250, 1996.
- [LSS99] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. On Efficiently Implementing SchemaSQL on a SQL Database System. In *VLDB*, pages 471–482, 1999.
- [O'B97] P. O'Brien. Making Java Objects Persistent. *Java Report*, 1(1):49–60, 1997.
- [Obj93] Object Design Inc. *ObjectStore - User Guide: DML. ObjectStore Release 3.0 for UNIX Systems*. Object Design Inc., 1993.
- [Obj99a] ObjectDomain. <http://www.objectdomain.com/>, 1999.
- [Obj99b] ObjectStore. <http://www.odi.com/objectstore/>, 1999.

- [RCL⁺99] E. A. Rundensteiner, K. T. Claypool, M. Li, L. Chen, X. Zhang, C. Natarajan, J. Jin, S. De Lima, and S. Weiner. SERF: ODMG-Based Generic Restructuring Facility. In *Demo Session Proceedings of SIGMOD'99*, pages 568–570, 1999.
- [Rea00] J. F. Roddick and L. Al-Jadir et al. Evolution and Change in Data Management - Issues and Directions. In *SIGMOD Record*, pages 21–25, March 2000.
- [Tec94] O₂ Technology. *O₂ Reference Manual, Version 4.5, Release November 1994*. O₂ Technology, Versailles, France, 1994.

A Metamodeling Approach to Evolution

Marie-Noëlle Terrasse

Laboratoire LE2I, Université de Bourgogne (France)
marie-noelle.terrasse@u-bourgogne.fr

Abstract. With the increasing complexity of systems being modeled, analysis & design move towards more and more abstract methodologies. Most of them rely on metamodeling tools that employ multi-view models and the four-layer metamodeling architecture. Our idea is to use the metamodeling approach to classify and to constraint the possible evolutions of an information system with the effect to improve both detection of evolution conflicts and disciplined reuse. Within the domain of UML metamodeling, a refinement of the metamodel-level classification is proposed that includes bases for defining a metric of the evolution (in terms of distance between metamodels).

1 Introduction

With the increasing complexity of systems being modeled, analysis & design methodologies rely on more and more abstract mechanisms that use multi-view models and metamodeling architectures [17]. As shown in Figure 1, multi-view models [3] refer to the principles of separation and combination of concerns [2,5,13] that are implemented through a metamodel (depicted by gray thick lines). Metamodeling tools generally refer to the four-layer architecture, in which each layer is an abstract description that provides a descriptive language to its lower layer, as well as evaluation and comparison criteria. In practice, the same descriptive language is used on different layers; this is called *loose metamodeling*. For example, the Unified Modeling Language (UML, [21]) is used as a descriptive language on the three uppermost layers of the OMG architecture, depicted in Figure 2, which includes:

- A model layer that is populated by a set of views through which the application is described. Those views are syntactically separate but semantically redundant.
- A metamodel layer that is populated by metamodels. They both determine the set of views that are necessary, and express constraints for integration of views in order to provide the application domain with specific multi-view models.
- A meta-metamodel layer that is populated by tools for evaluation and alignment of metamodels. The Meta Object Facility (MOF, [20]) of the OMG belongs to this layer.

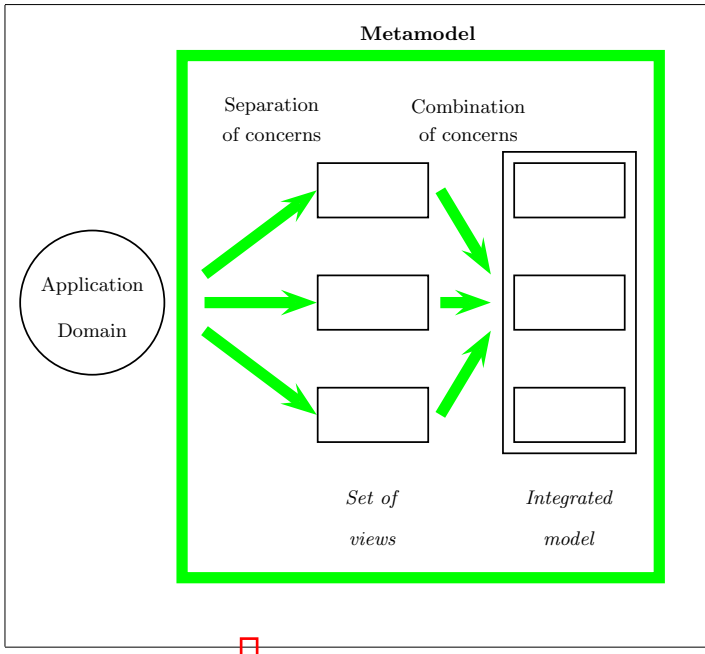


Fig. 1. Multi-view model defined by a metamodel

Our approach is to define different types of evolution for an information system by using each layer as a constraint for the evolution of its lower layer (Section 2). We focus on the metamodel layer in order to refine our classification of metamodel-level evolution (Sections 3 and 4). The ongoing work is presented in the conclusion (Section 5).

2 Metamodeling and Evolution

For long time (see, for example, Orion [1]), schema evolution, i.e. the ability to dynamically modify a schema (including a definition of a semantics for schema evolution as well as implementation issues) has been a major requirement for Object-Oriented databases. In the domain of information systems, it is possible to use the whole of the metamodeling architecture and to identify a type of evolution for each layer:

- **Data-level evolution** is the result of system activities. This evolution is required to be consistent with the behavior of the system (Activity and State diagrams), the interactions of its components (Sequence and Collaboration diagrams), the behavior of users (Use Case diagram), etc.
- **Model-level evolution** is the result of evolution of the application itself. This evolution is required to be consistent with all constraints expressed in the metamodel: any modification of one particular model \mathcal{M} must propagate

- to every model related to \mathcal{M} within the metamodel. For example, the class diagram may be extended to describe a new part of the application; this extension must propagate to behavioral model(s).
- **Metamodel-level evolution** is the result of evolution of the application domain. For example, it may be useful to introduce extra views that describe a new aspect of systems, e.g., to add a Use-Case diagram in order to take into account different types of users that are to be offered different functionalities.
 - **Meta-metamodel-level evolution** is the result of evolution of the modeling paradigm, i.e., the “filter” through which the real world is viewed. For example, the underlying Boolean logic may be changed to a modal logic.

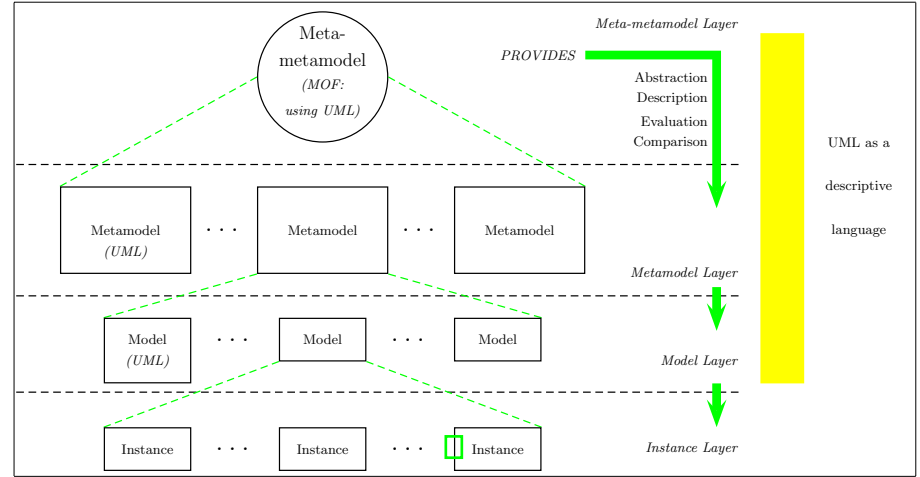


Fig. 2. The OMG's four-layer architecture

Similarly, we can partition the sets of invariant properties and rules that guard evolution [1] into several levels of abstraction. Obviously, most abstract controls (at metamodel and meta-metamodel levels) allow high-level supervision of the evolution. We focus on those two uppermost layers of the metamodeling architecture (Section 3). For this purpose, we define a modeling paradigm as a requirement, i.e., as the specification of the constraints under which the target system will be modeled. We then deduce a corresponding sub-classification of metamodel-level evolution and give the bases for measurement of evolution in terms of distances between metamodels (Section 4). By itself, such a distance provides an evaluation of the gap between the original and the target information systems. Furthermore, since the computation is carried out by comparing components of the original and the target metamodels, such a distance provides the evolution process with a semantical background.

3 Our Approach to Metamodeling with UML

Our architecture relies upon the notion of *modeling paradigms* that are abstract descriptions of sets of requirements under which systems are being modeled. They can use both natural and formal languages. Naturally, a choice of a language (e.g., a choice of a particular logic) may make expressing some modeling paradigms impossible. In any case, our objective is to have a wide-purpose architecture that minimizes the number of inexpressible modeling paradigms.

Towards this goal, we propose to build a two-fold structure consisting of the two uppermost layers of the metamodeling architecture, in which:

- A part of the meta-metamodel layer, denoted by $Restrict_{MP}$, contains all modeling paradigms that are compliant with the UML expressiveness. $Restrict_{MP}$ is organized into a semi-lattice of modeling paradigms that are partially ordered by a subsumption relationship denoted by \preceq .
- In the above set $Restrict_{MP}$, an instantiation function, denoted by \mathcal{E} , associates a modeling paradigm with a specialization of the UML metamodel (using UML's tailoring mechanisms [12]).
- At the metamodel level, the range of \mathcal{E} , denoted by $\mathcal{E}(Restrict_{MP})$, is populated by specializations of the UML metamodel. $\mathcal{E}(Restrict_{MP})$ is organized as an inheritance hierarchy of metamodels which mirrors –more or less accurately– the semi-lattice of modeling paradigms.

We are convinced that this two-fold structure can provide an efficient support for management –at high level of abstraction– of information systems: it is possible to define formal operations on formalized metamodels. In order to fully play this role, our structure must guarantee correctness of formalized metamodels and provide corresponding formal tools. Our meta-metamodel and metamodel layers are described in Sections 3.1 and 3.2.

3.1 Meta-metamodel Layer

Our meta-metamodel layer comprises modeling paradigms that refer to a set of concepts (objects, classes, time and space models, etc.) and languages (English language, logic, set theory, etc.) that are well known but may be ambiguous. They are supposed to describe –as precisely as possible– the subset of concepts that will be used to express the semantics of the real world. We define modeling paradigms the following way:

Definition 1 (Description of modeling paradigms)

A modeling paradigm mp is described in terms of English language, logic and set theory. Its description comprises two sets, $\mathcal{E}l(mp)$ and $\mathcal{C}(mp)$. The set $\mathcal{E}l(mp)$ contains descriptions of elementary concepts, while the set $\mathcal{C}(mp)$ contains constraints among the concepts of $\mathcal{E}l(mp)$.

□

Let us denote by gmp a general modeling paradigm that corresponds to the OO modeling paradigm, as expressed in the UML approach [21]. The modeling paradigm gmp is described by:

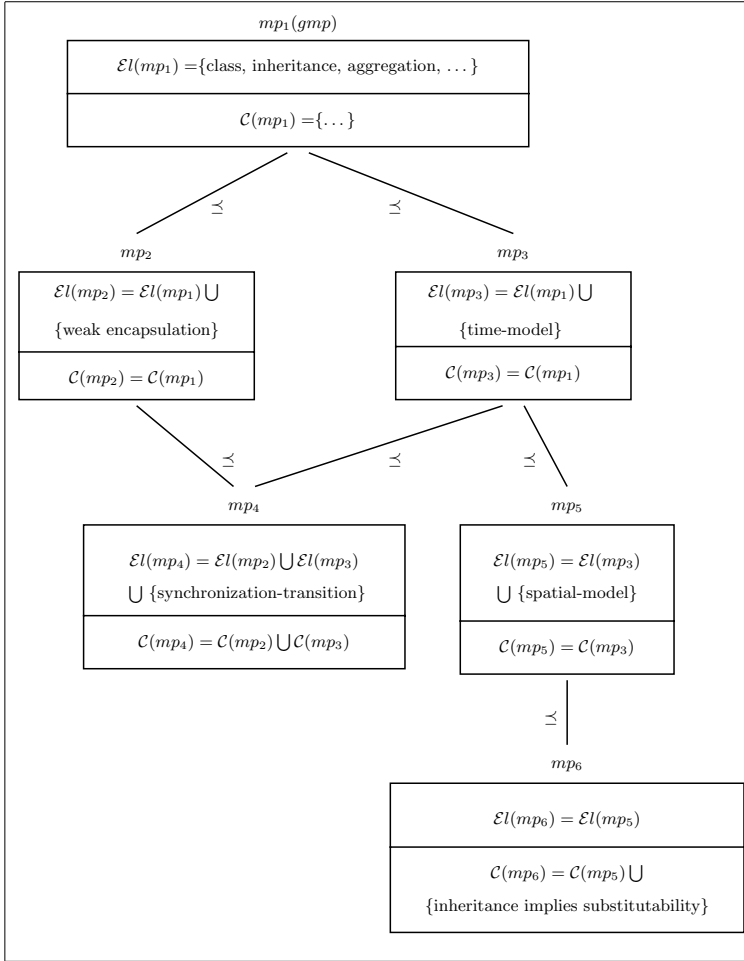


Fig. 3. A Poset of Modeling Paradigms

- A set $\mathcal{El}(gmp)$ that contains descriptions (in English, logic, or set theory) of objects with identity, state, and behavior; class; generalization; inheritance; aggregation; etc.
- A set $\mathcal{C}(gmp)$ that contains rules (in English, logic, or set theory expressions) like “each object belongs to a class” or “generalization implies substitutability”, etc.

See Figure 3 for examples of modeling paradigm descriptions. In order to make this figure readable, the descriptions are limited to sets of concepts, and constraints are expressed in natural language. The description of modeling paradigm mp_1 includes usual OO concepts (such as class, inheritance, aggregation), as well as constraints. Modeling paradigm mp_2 is built from mp_1 by adding a new concept for weak encapsulation in order to deal with inheritance

anomaly in synchronization [8]. Modeling paradigm mp_6 is built from mp_5 by adding a new constraint “*Inheritance implies substitutability*”.

As shown in the previous examples, modeling paradigms may use a various number of concepts. A minimal OO modeling paradigm may have only one concept of class, while *gmp* distinguishes several kinds of classes (e.g., an abstract class or an implementation class). Similarly, two modeling paradigms using the same concept may define that concept with more or fewer details and constraints. Thus, we define a partial ordering relationship between modeling paradigms:

Definition 2 (Subsumption of modeling paradigms)

We say that a modeling paradigm mp_1 is *subsumed by* a modeling paradigm mp_2 , which is denoted by $mp_1 \preceq mp_2$, if both of the following conditions are satisfied:

Extended inclusion of elements. Each element of $\mathcal{El}(mp_2)$ is either a member of $\mathcal{El}(mp_1)$ or a generalization of an element of $\mathcal{El}(mp_1)$, where a generalized element may have fewer features than its specialized element has.

Subsumption of constraints. Using $\mathcal{C}(mp_1)$ as a hypothesis, it is possible to prove that each constraint of $\mathcal{C}(mp_2)$ holds.

Two modeling paradigms that are related by \preceq or by the inverse relationship *subsumes* (denoted by \succeq) are said to be *comparable*.

□

□

In our architecture, the ordering relationship between modeling paradigms may either be given by users (when they explicitly build a new modeling paradigm as a special case of an existing one) or evaluated by the system (by extended inclusion of sets of concepts and subsumption of constraints).

Figure 3 presents ordered modeling paradigms that correspond either to added constraints (mp_6) or to extended inclusion of concepts (mp_4 [7,8]; mp_3 [10]; and mp_5 are built by adding new concepts while mp_2 [7,14] is built by specializing an existing concept).

The subsumption of modeling paradigms is a partial ordering relationship¹. Thus, our meta-metamodel layer may be structured as a poset of modeling paradigms². Since all ordering relationships are not necessarily given, we use the following closures:

Definition 3 (Closures)

We call the set of links that must be given a *non-trivial set*. From the set of non-trivial links, computation of the reflexive-transitive³ closure may be carried

¹ Ordering relationship in the mathematical sense, i.e., a relation that is reflexive, anti-symmetric and transitive.

² Note that neither reflexive or transitive links are represented in Figure 3.

³ A reflexive closure encompasses ordering links from each modeling paradigm to itself. Analogously, transitive closure encompasses a direct link from an initial modeling paradigm to a final one of each sequence of ordering links.

out by using classical algorithms. We call the set of all links corresponding to such a closure of the inverse relationship \succeq an *inverse closure*.

The total closure, denoted by (S, \preceq) , includes links from both reflexive-transitive and inverse closures, and thus contains all pairs of comparable modeling paradigms.

□

We are interested in all modeling paradigms that are subsumed by *gmp*. Let us call by $Restrict_{MP}$ the set of such modeling paradigms. The subsumption of modeling paradigms defines a meet-semi-lattice [6] on $Restrict_{MP}$, i.e. a set with a partial ordering relationship such that every pair of elements of $Restrict_{MP}$ has a least-upper-bound in $Restrict_{MP}$.

Since our purpose is to make comparisons of modeling paradigms as easy as possible, it is important to have an ordering relationship for which most of the modeling paradigms are comparable. Thus, we define several properties in order to evaluate the quality of a partial order.

Definition 4 (*Evaluation of a partial order*)

In this definition we introduce coverage as a global evaluation of a partial order. We define a sub-poset, as well as two properties that can apply to sub-posets in order to evaluate local qualities of a partial order.

Coverage of partial order. The coverage of a partial order \preceq on a set S of modeling paradigms, denoted by $Cov(S, \preceq)$, is a real value lying between 0 and 1 that indicates the ratio between the number of pairs of comparable modeling paradigms in S and the number of all possible pairs, i.e. the cardinality of S^2 :

$$Cov(S, \preceq) = \frac{|\overline{(S, \preceq)}|}{|S^2|}$$

Depth of a poset. The depth of a poset induced on a set S by \preceq , denoted by $\mathcal{D}p(S, \preceq)$, is the length of longest sequences of ordered modeling paradigms:

$$\mathcal{D}p(S, \preceq) = \text{Max}\{n \mid \exists (s_1, s_2, \dots, s_n) \in S^n \text{ with } s_1 \preceq s_2 \preceq \dots \preceq s_n\}$$

Sub-poset. Consider a poset S (i.e. a set of modeling paradigms with a partial order \preceq) and a subset *sub* of S . The restriction of \preceq on *sub* is a partial order on *sub*. Since *sub* itself is a poset, we call it a sub-poset of S .

Independent sub-poset. Consider a poset S and a sub-poset *sub* of S . We call any link (corresponding to either \preceq or \succeq relationships) between a modeling paradigm of *sub* and a modeling paradigm of $S \setminus sub$ ⁴ an *external link* of *sub*. In a poset including the modeling paradigm *gmp*, a sub-poset *sub* is said to be independent if *gmp* is the only modeling paradigm to which *sub* is externally linked.

⁴ As defined in standard set theory, $S \setminus sub$ denotes the set of elements of S that do not belong to *sub*.

Note that coverage, depth and independence have same meaning when applied to semi-lattices instead of posets.

□

The number and depth of independent sub-posets provide an approximation of the coverage since many deep independent sub-posets tend to make the coverage low. If we were working with general modeling paradigms, the coverage would be low. In our architecture, each additional constraint that applies to *gmp* corresponds to one of the potential variations between members of the UML family of languages. All possible ambiguous elements, as well as the set of possible choices for each element, are well known. It is thus possible to predict the depth of the lattice. Furthermore, independent sub-semi lattices are generated by fundamental choices when defining an extension for a new domain. Thus, the number of independent sub-semi lattices should be close to the number of different domains that are described.

3.2 Metamodel Layer

Our objective is to build our metamodel layer as a mirror of the semi-lattice of modeling paradigms: the generic modeling paradigm *gmp* is instantiated into the UML metamodel itself, and any other modeling paradigm is instantiated into a specialization of the UML metamodel.

□

Specializations of the UML metamodel. Many examples of application-domain specific metamodels are available in the UML literature [4,7,10,11]. They use the tailoring mechanisms (constraints, tag values and stereotypes) of UML [12]. Let us present two of them:

- Herrero & al. [7] describe in detail an extension of UML metamodel for synchronization. They define a stereotype of class that allows weak encapsulation for non-functional properties of behavior descriptions. This stereotype replaces standard UML's classes. They also define –as a stereotype– a special kind of statechart with a new concept of synchronization-transition that encompasses two extra actions (pre- and post-actions).
- Robbins & al. [11] extend UML for ADL (architecture description language). They define the concept of message specification in ADL's language C2, augmenting UML's concept of operation with both a tagged value differentiating notifications from requests, and a constraint stipulating that no result is allowed.

Instantiation of a modeling paradigm. We propose to use similar mechanisms in order to instantiate a modeling paradigm into a specialized metamodel. More precisely, each modeling paradigm is instantiated as a specialization of either the UML metamodel itself or a specialization of its subsuming modeling paradigm(s):

Definition 5 (*Instantiation of a modeling paradigm*)

Consider a modeling paradigm *mp*. Each concept of $\mathcal{E}l(mp)$ is instantiated, by

the instantiation function \mathcal{E} , into one or more components of the UML language that are linked (or made precise) by some constraints. Additional constraints are generated from the set of constraints $\mathcal{C}(mp)$. Thus, we assume that mp 's corresponding metamodel $mm = \mathcal{E}(mp)$ is described by a set of components $Comp(mm)$ and a set of constraints $Constr(mm)$.

□

Furthermore, we require that our instantiation complies with the ordering of modeling paradigms so that our metamodel layer can be a mirror of the meta-metamodel layer: each modeling paradigm is mirrored by its instantiation as a metamodel, each ordering relationship between modeling paradigms is mirrored by an inheritance relationship between metamodels:

Rule 1 (Full compliance of instantiation)

If a modeling paradigm mp is subsumed by a modeling paradigm mp' , then mp' must be instantiated as a specialization of the instantiation of mp .

□

Such a full compliance makes the use of multiple inheritance mandatory; see [16] for an alternative to multiple inheritance.

Figure 4 presents such a perfect mirroring for the semi-lattice that was used in Figure 3. Instantiations are represented by large grey arrows.

We assume that mp_1 is the general modeling paradigm gmp . It is instantiated as the UML metamodel itself. We have

$$Comp(UMLMetamodel) = \{class, aggregation, \dots\}$$

See Herrero & al. [7] for an example. Each modeling paradigm is instantiated as a specialization of the UML metamodel, e.g.

$$Comp(mm_2) = \{\ll nf_class \gg, aggregation, \dots\}$$

where $\ll nf_class \gg$ is a specialization of class –defined as a stereotype– that distinguishes two kinds of properties (functional and non-functional) within behavioral descriptions.

Analogously, a stereotype $\ll synchronization_transition \gg$ of UML's transition describes non-functional properties of behavior in the context of synchronization. Thus, we have

$$Comp(mm_4) = Comp(mm_2) \cup \{\ll synchronization_transition \gg\} \cup Comp'(mm_4)$$

where $Comp'(mm_4)$ contains time-model components, which are not described in detail here.

Each ordering relationship at the meta-metamodel level corresponds to exactly one inheritance relationship at the metamodel level. Multiple inheritance is necessary for the mm_4 metamodel. We have a perfect mirror in terms of elements as well as in terms of relationships: our instantiation fully complies with the structure of the semi-lattice of modeling paradigms.

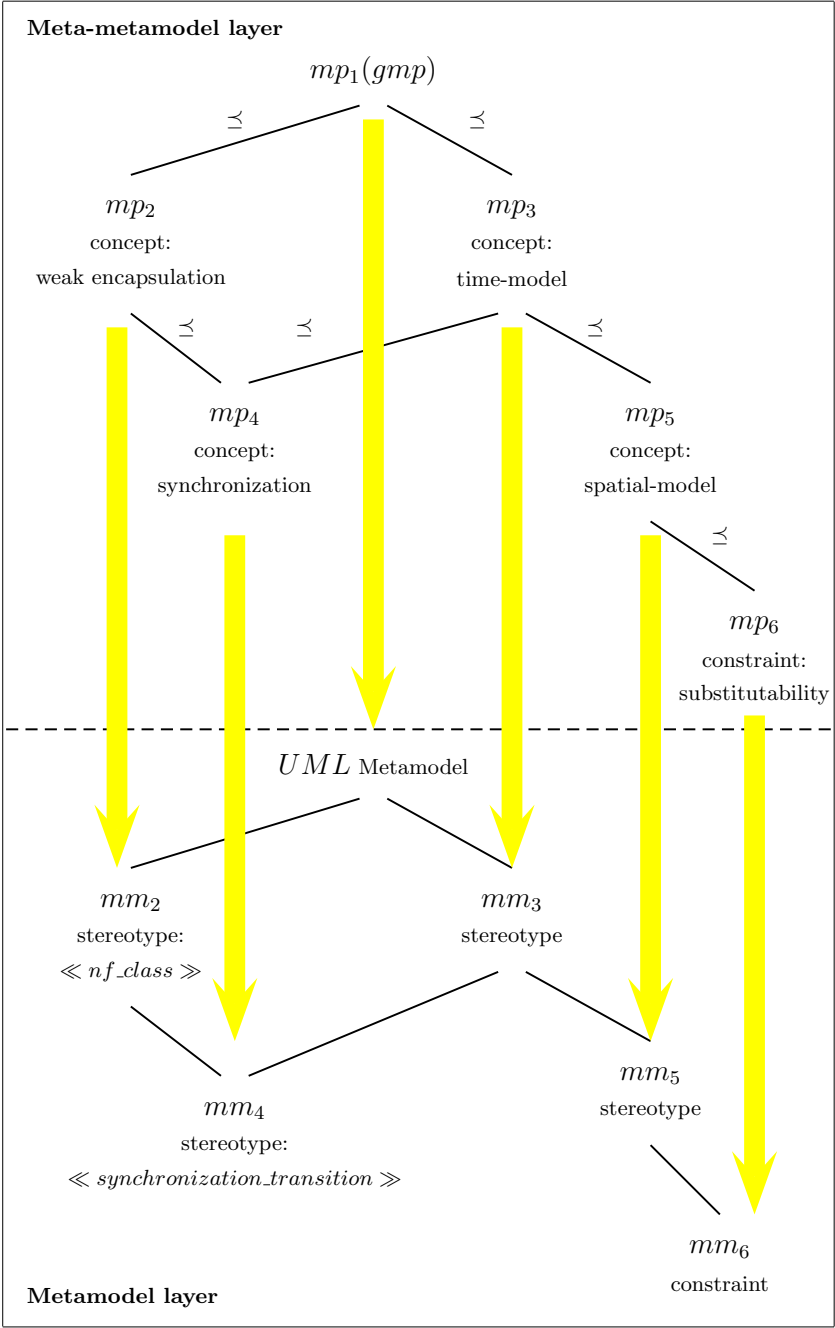


Fig. 4. Mirroring of a Semi-lattice of Modeling Paradigms

4 Metamodel-Level Evolution

In Section 4.1, we use our metamodeling architecture to determine different types of metamodel-level evolution. When evolving from an initial information system to a target one, we compare the positions of the corresponding metamodels within the inheritance hierarchy of metamodels. Section 4.2 develops our strategy for defining distance between metamodels corresponding to each potential type of evolution.

4.1 Classification of Evolution

Given an initial metamodel denoted by \mathcal{MM}_{init} and a target metamodel denoted by \mathcal{MM}_{targ} , let us define the following cases (examples refer to Figure 4):

- **Restriction:** The initial paradigm is a weaker requirement than the target one. Thus \mathcal{MM}_{init} is one of the super-classes of \mathcal{MM}_{targ} , e.g., evolving from mm_3 to mm_6 is a restriction. It is possible to evaluate the distance between the two formal metamodels.
- **Relaxation:** The initial paradigm is a stronger requirement than the target one. Thus \mathcal{MM}_{init} is one of the subclasses of \mathcal{MM}_{targ} , e.g., evolving from mm_6 to mm_3 is a relaxation. It is possible to evaluate the distance between the two formal metamodels.
- **Evolution with a formal ancestor:** The initial and target paradigms have a formalized common part corresponding to a common metamodel ancestor – which can't be the UML metamodel itself-. For example, evolving from mm_4 to mm_6 is an evolution with a formal ancestor mm_3 . The distance between metamodels can be evaluated since both the common and the specific parts are formalized.
- **Evolution without a formal ancestor:** UML metamodel is the only common ancestor to \mathcal{MM}_{init} and \mathcal{MM}_{targ} . For example, evolving from mm_2 to mm_5 is an evolution without a formal ancestor. In this case, the inheritance hierarchy does not permit to directly define the distance between the initial and target metamodels. However, it is possible to build an ad-hoc ancestor by going up to the meta-metamodel level in order to define an intermediate modeling paradigm \mathcal{IMP} that encompasses common features of \mathcal{MP}_{init} and \mathcal{MP}_{targ} . If \mathcal{IMP} is an unambiguous modeling paradigm, then $\mathcal{E}(\mathcal{IMP})$ is defined and may be used as a formal ancestor.

From the inheritance hierarchy of metamodels, we have refined the classification of metamodel-level evolutions. We use distance between metamodels in two different configurations which encompass all possible relative locations of the initial and target metamodels. The first configuration is a *sequence configuration* corresponding to restrictions or relaxations. In this case, the considered metamodels belong to the same branch of our inheritance hierarchy. The second configuration is a *configuration with a pivot* in which a common ancestor, called *pivot*, encompasses common parts of the metamodels to be compared. Next section develops two basic evaluations of distance corresponding to those configurations.

4.2 Bases for Distance between Formal Metamodels

Our idea for defining distance between formal metamodels is to build the distance as a weighted sum of elementary distances between similar elements of metamodels (i.e., similar components and similar constraints). Since we work within a formalized context, it is possible to exactly determine the set of potential components of the metamodel as well as its set of constraints. Since a definition of distance at such a fine granularity (element by element) is difficult to implement with sufficient efficiency, we propose to partition the sets of components and constraints into subsets on which distance can be defined globally, i.e. by using the same criterion. The partition process relies upon our two-fold structure of modeling paradigms and metamodels. In this way, distance between metamodels is defined as a weighted sum of elementary distances between subsets of components and constraints.

Sections 4.2 and 5 detail the construction of distance between two modeling paradigms in a sequence configuration. Section 5 explains what is specific in the case of a configuration with a pivot. Section 5 summarizes main features of our distance.

Distance between components in a sequence configuration. Let us consider two metamodels mm_1 and mm_2 and their corresponding modeling paradigms mp_1 and mp_2 , respectively. We assume that mm_1 is a direct heir of mm_2 . Due to our rule of the full compliance, this means that $mp_1 \preceq mp_2$. We use our definition of subsumption of modeling paradigms to partition the sets of components of mm_1 and mm_2 .

Rule 2 (Partition of concepts, partition of components)

The partitioning of concepts of mp_1 and mp_2 produces three subsets of concepts:

- Common concepts:
A set $Com(mp_1, mp_2)$ contains all concepts that belong to $\mathcal{El}(mp_1) \cap \mathcal{El}(mp_2)$.
- Specialized concepts:
A set $Spe(mp_1, mp_2)$ contains all concepts of mp_1 that specialize a concept of mp_2 .
- New elements:
A set $New(mp_1, mp_2)$ contains all other concepts of mp_1 .

Using our instantiation function \mathcal{E} , we can also generate a partition of the components of mm_1 and mm_2 metamodels. Let us denote by $Com_{UML}(mm_1, mm_2)$, $Spe_{UML}(mm_1, mm_2)$, and $New_{UML}(mm_1, mm_2)$, respectively, subsets of components corresponding to the three above subsets of concepts.

□

In the following, we focus on criteria that can be used to define elementary distances associated with each subset:

- Common components share the same concept. The only difference may be in the way the shared concept has been translated by \mathcal{E} into mm_1 and mm_2 metamodels. In many cases, this distance should be ignored. We denote by $D_{Com}(mm_1, mm_2)$ the corresponding distance.
- Specialized components share, through more general components, the same concept: we work with pairs of components corresponding to the concept itself and its successor in the semi-lattice of paradigms. For each pair of general-specialized components, evaluation of distance must take into account both variations due to \mathcal{E} -instantiation and the variations introduced by added features. We denote by $D_{Spe}(mm_1, mm_2)$ the corresponding distance.
- New components are more tricky. They are added to mm_1 metamodel and we have to evaluate both the cost of the concept itself and the way this concept is translated. For example, many designers try to avoid using stereotypes because they introduce high-cost distortions [11]. We denote by $D_{New}(mm_1, mm_2)$ the corresponding distance.

Finally, the distance between two metamodels, in terms of their components, is defined as a weighted sum:

$$D_{Comp}(mm_1, mm_2) = \alpha D_{Com}(mm_1, mm_2) + \beta D_{Spe}(mm_1, mm_2) + \delta D_{New}(mm_1, mm_2)$$

Weights (denoted by Greek letters) may be equal to zero if their corresponding elementary distances are not significant. Different strategies for choosing optimum weights are to be explored in the short-term ongoing work.

For example, if we wish to evaluate the distance between metamodels mm_4 and mm_1 ⁵ of Figure 4, we have to build:

- At the meta-metamodel level, the set of common concepts:

$$Com(mp_1, mp_4) = \{class, inheritance, aggregation\}$$

- At the meta-metamodel level, the set of specialized concepts:

$$Spe(mp_1, mp_4) = \{weak_encapsulation\}$$

- At the meta-metamodel level, the set of new concepts:

$$New(mp_1, mp_4) = \{synchronization_transition\}$$

- At the metamodel level, the pairs of general-specialized components of UML:

$$Spe_{UML}(mm_1, mm_4) = \{(class, stereotype \ll nf_class \gg)\}$$

the corresponding distance must evaluate the distortion between the original element (*class*) and the stereotype ($\ll nf_class \gg$).

⁵ Corresponding to modeling paradigms mp_4 and mp_1 , respectively. Note that we need to assume that mp_1 is not the (ambiguous) general modeling paradigm gmp so that mm_1 may be a formal metamodel.

- At the metamodel level, the set of new UML components:

$$New_{UML}(mm_1, mm_4) = \{\textit{stereotype} \ll \textit{synchronization_transition} \gg\}$$

The corresponding distance depends on the importance given to the concept of *synchronization_transition* itself, as well as to its translation.

Global Distance in a Sequence Configuration. Analogously to the above, we use the definition of the subsumption of modeling paradigms to determine a partition of the constraints of mm_1 and mm_2 into subsets that correspond to a uniform evaluation of distance.

Rule 3 (Partition of constraints)

The partition of constraints of the previous modeling paradigms mp_1 and mp_2 produces three subsets of constraints:

- Shared constraints:

A set $Shar(mp_1, mp_2)$ contains all constraints that belong to $\mathcal{C}(mp_1) \cap \mathcal{C}(mp_2)$. □

- Deduced constraints:

A set $Ded(mp_1, mp_2)$ contains all other constraints that are used to deduce non-shared constraints of mp_1 (by definition of our subsumption).

- Added constraints:

A set $Add(mp_1, mp_2)$ contains all other constraints of mp_2 .

Because we use formalized constraints (OCL [18]) in the corresponding meta-models, we assume that the distance due to shared constraints, as well as to deduced constraints, is not significant.

□

We denote by $D_{Add}(mm_1, mm_2)$ the distance between mm_1 and mm_2 that corresponds to added constraints. We propose to determine the significance of an added constraint by evaluating –in the context of the concerned application’s domain– the number of potential components (or potential groups of components) that are excluded by the constraint.

Then, the global distance of mm_1 and mm_2 is defined as a weighted sum of the previous distances:

$$D_{seq}(mm_1, mm_2) = \lambda D_{Comp}(mm_1, mm_2) + \mu D_{Add}(mm_1, mm_2)$$

Distance for a configuration with a pivot . In order to evaluate the distance of components in a configuration with a pivot, we define a specific partition of concepts and components. Such a partition encompasses common and new concepts defined previously. The set of specialized concepts has to be refined to comply with the configuration with a pivot. Let us consider $mm_1 = \mathcal{E}(mp_1)$ and $mm_2 = \mathcal{E}(mp_2)$, two metamodels whose distance has to be evaluated, and their common ancestor $mp_0 = \mathcal{E}(mp_0)$.

Rule 4 (*Partition of concepts and components with a pivot*)

The partition of concepts for two modeling paradigms mp_1 and mp_2 with a common ancestor mp_0 encompasses four subsets:

- Common concepts:
A set $Com(mp_1, mp_2)$ contains all concepts of mp_0 that also belong to $\mathcal{El}(mp_1) \cap \mathcal{El}(mp_2)$.
- Bi-specialized concepts:
A set $Bi(mp_1, mp_2)$ contains all concepts of mp_0 that are specialized in mp_1 and mp_2 .
- Uni-specialized concepts:
A set $Uni(mp_1, mp_2)$ contains all concepts of mp_0 that are specialized by either a concept of mp_1 or (exclusively) by a concept of mp_2 .
- New elements:
A set $New(mp_1, mp_2)$ contains all other concepts of mp_1 and mp_2 .

This partition induces a partition of UML components of the corresponding metamodels mm_1 and mm_2 into common, bi-specialized, uni-specialized, and new components.

□

Elementary distances for common, uni-specialized and new components can be evaluated by using criteria such as those defined for a sequence configuration. We propose to view bi-specialized components as a combination of two uni-specializations (from mm_0 to mm_1 and from mm_0 to mm_2), and to evaluate their distance as the sum of distances of the corresponding uni-specializations.

The distance between components in a configuration with a pivot is defined as a sum of those four specific distances:

$$D_{Comp}(mm_1, mm_2) = \alpha D_{Com}(mm_1, mm_2) + \beta_1 D_{Uni}(mm_1, mm_2) + \beta_2 D_{Bi}(mm_1, mm_2) + \delta D_{New}(mm_1, mm_2)$$

Similarly, we have to extend the definition of the set of added constraints. We denote by $D_{1Add}(mm_1, mm_2)$ and $D_{2Add}(mm_1, mm_2)$ the distances corresponding to constraints added to mm_1 and mm_2 , respectively.

Then, the global distance between mm_1 and mm_2 is defined as a weighted sum of the previous distances:

$$D_{piv}(mm_1, mm_2) = \lambda D_{Comp}(mm_1, mm_2) + \mu_1 D_{1Add}(mm_1, mm_2) + \mu_2 D_{2Add}(mm_1, mm_2)$$

Main features of our distance. The strategy that we propose for defining distance between formalized metamodels has two main features.

First, the partition of components and constraints into subsets that can be subjected to uniform criteria allows a definition of distance at a medium granularity. Note that –except in the case of restriction and relaxation– our partition

relies upon the inheritance hierarchy (through the determination of an ancestor). Thus, the quality of the inheritance hierarchy determines the accuracy of distance [15].

Second, elementary distances, as well as weights used for their combination, can be fine-tuned in the context of a specific application domain. This requires good knowledge of the application domain from the designer of the distance. Such a requirement is consistent with the fact that the distance is defined at the metamodel level.

5 Conclusion

By using the metamodeling point of view, we have proposed a classification of evolutions of an information system (from instance-level evolution to meta-metamodel-level evolution) and established the constraints that restrict this evolution. Within the particular domain of an UML-modeling architecture, we have refined the classification of the metamodel-level evolution, defined a criterion to identify “measurable” types of evolution, and established bases for defining distances between metamodels.

Furthermore, the structure that we propose reveals dependencies among different versions of information systems which are organized into a hierarchy of more and more restricted metamodels that are all formalized: each sub-hierarchy of metamodels corresponds to a particular application domain. By using this structure, we can cope with two issues of evolution [9]: (a) evolution conflicts that result into an inconsistent model can be detected (by using a formal toolbox) since the underlying metamodel is formalized, (b) disciplined reuse is made easier by using the inheritance hierarchy since each specific domain of application fits a sub-hierarchy.

The ongoing work encompasses both technical and validation aspects. On the technical side, we plan to make the formalization process explicit by defining the \mathcal{E} -translation. We plan to find out whether the guarding condition that guards the \mathcal{E} -translation may be weakened. We have proposed a sample of guarding conditions [16] that have to be tested in different contexts. In order to validate our approach, we will develop a sample of domain-specific distances. Each of them is to be validated through domain expertise rules (see Wedemeijer [19]). Furthermore, we expect that such a sample will be the basis for determining more general criteria.

Acknowledgment. The work described in this paper has been carried out at the National Institute of Standard and Technology. It is the result of fruitful discussions with Christopher E. Dabrowski, Leonard J. Gallager, Lisa J. Carnahan, and George Becker there, as well as with Marinette Savonnet at the University of Burgundy. Many thanks for their questions and remarks that have contributed to substantial improvement of this paper.

References

1. Jay BANERJEE, Won KIM, Hyoung-Joo KIM, and Henry F. KORTH. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the Annual Conference on Management of Data*, pages 311–322. ACM SIGMOD, May 1997.
2. Jean BEZIVIN. On Different Interoperability Modes in Software Engineering: the Case of Modeling Activities at OMG. In *Proceedings of Software Engineering'98*, Paris, France, December 1998.
3. Ruth BREU, Radu GROSU, Franz HUBER, Bernhard RUMPE, and Wolfgang SCHWERIN. Systems, Views and Models of UML. In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, 1998. Available at URL <http://www.cs.york.ac.uk/puml>.
4. M.A. DE MIGUEL, A. ALONSO, and J.A. DE LA PUENTE. Object-Oriented Design of Real-Time Systems with Stereotypes. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, pages 216–223. IEEE, June 1997.
5. Robert GEISLER, Marcus KLAR, and Claudia PONS. Dimensions and Dichotomy in Metamodeling. In *3rd BCS-FACS Northern Formal Methods Workshop*, UK, September 1998. Available at URL <http://www-lifia.info.unlp.edu.ar/~cpons>.
6. George GRÄTZER. *Lattice Theory, First Concepts and Distributive Lattices*. W.H. Freeman, 1971. ISBN 0-7167-0442-0.
7. José Luis HERRERO, Fernando SANCHEZ, Fabiola LUCIO, and Miguel Toro BONILLA. Changing UML Metamodel in Order to Represent Concern Separation. ECOOP'00 Workshop 14 on Defining a Precise Semantics for UML, Sophia Antipolis, France, June 2000.
8. S. MATSUOKA, K. TAURA, and A. YONEZAWA. Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages. In *Proceedings OOPSLA'93 Workshop on Object-Oriented Behavioral Semantics*, pages 109–126. ACM, 1993.
9. Tom MENS, Carine LUCAS, and Patrick STEYAERT. Supporting Disciplined Reuse and Evolution of UML Models. In *The Unified Modeling Language - UML'98: Beyond the Notation*, pages 378–392. Springer, LNCS 1618, 1998.
10. Rosanne PRICE, Bala SRINIVASAN, and Kotagiri RAMAMOCHANARAO. Extending the Unified Modeling Language to Support Spatiotemporal Applications. In C. Mingins and B. Meyer, editors, *Proceedings of TOOLS 32, Conference on Technology of Object-Oriented Languages and Systems*, pages 163–174. IEEE, November 1999.
11. Jason E. ROBBINS, Nemađ MEDVIDOVIC, David F. REDMILES, and David S. ROSENBLUM. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 209–218. IEEE, April 1998.
12. James RUMBAUGH, Ivar JACOBSON, and Grady BOOCH. *The Unified Modeling Language - Reference Manual*. Addison-Wesley, 1998. ISBN 0-201-30998-X.
13. Bernhard SCHÄTZ and Franz HUBER. Integrating Formal Description Techniques. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - Formal Methods*, volume 2 of *LNCS 1709*, pages 1206–1225. Springer Verlag, September 1999. Available at URL <http://www4.informatik.tu-muenchen.de/papers>.
14. Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Available at URL www.objecttime.com/otl/technical/umlrt.html, 1998.
15. Marie-Noëlle TERRASSE. Using UML-Metamodeling for Interoperable Geographic Information Systems. Technical Report 00-6, Laboratoire LE2I, Université de Bourgogne, France, 2000.

16. Marie-Noëlle TERRASSE and Marinette SAVONNET. Formalization of the UML Metamodel: An Approach Based Upon the Four-Layer Metamodeling Architecture. ECOOP'00 Workshop 14 on Defining a Precise Semantics for UML, Sophia Antipolis, France, June 2000.
17. Marie-Noëlle TERRASSE and Marinette SAVONNET. Metamodeling with the UML: An Approach to the Formalization of the UML Metamodel. In *5th CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, EMMSAD'00*, pages E1–E10, June 2000. Stockholm, Sweden.
18. Jos WARMER and Annekke KLEPPE. *The Object Constraint Language - Precise Modeling with UML*. Addison-Wesley, 1999. ISBN 0-201-37940-6.
19. L. WEDEMEIJER. Defining Metrics for Conceptual Schema Evolution. In H. Balsters, B. De Brock, and S. Conrad, editors, *Database schema Evolution and Meta-Modeling — 9th International Workshop on Foundations of Models and Languages for Data and Objects (FOMLADO/DEMM 2000)*, Dagstuhl, Germany, Lecture Notes in Computer Science Vol. 2065, pages 219–243, Springer-Verlag, 2001.
20. *Meta Object Facility (MOF) Specification*, September 1999. Version 1.3, Available at URL www.omg.org.
21. *OMG Unified Modeling Language Specification*, March 2000. Version 1.3, Available at URL <http://www.omg.org>.

Defining Metrics for Conceptual Schema Evolution

Lex Wedemeijer

ABP Pensioenen, The Netherlands

L.Wedemeijer@wxs.nl

Abstract. It is generally believed that a well-designed Conceptual Schema will remain stable over time. However, current literature rarely addresses how such stability should be observed and measured in the operational business environment with evolving information needs and database structures. This paper sets up a framework for stability of conceptual schemas and proceeds to develop a set of metrics from it. The metrics are based on straightforward measurements of conceptual features. The validity of the set of metrics is argued here from theory, operational validity may be demonstrated by a longitudinal case study into the evolution of conceptual schemas. The main contribution of this paper is the realization that the measurement of conceptual schema stability is an essential step for understanding and improving current theories and best-practices for designing high-quality schemas that will stand the test of time.



1 Introduction

According to the 3-schema architecture, a well-designed Conceptual Schema (CS) satisfies many quality requirements [5,30,35]. It is the task of the designer to meet these requirements in the best possible way. In particular, the CS is required to be stable enough to support a long-term systems lifetime and be flexible enough to meet future information demands. Many design strategies exist that claim to improve the flexibility of the CS design. Why they should enhance flexibility is often explained, sometimes demonstrated, but rarely proven by actual business cases. A designer that wants to prepare the CS for future changes, must trust to experience and to state-of-the-art design practices, there is no way to pick the ‘best’ design strategy for a particular business case at hand. Also, a cry for wholesale flexibility of CSs is not a very specific requirement that designers can meet with:

- flexibility can only be established ‘on the fly’. A potential for change can only become apparent when a structural change occurs, and not when discussing a new schema
- there is no distinction between structural changes that ought to be accommodated by the flexibility in the CS design, and those that are beyond the desired flexibility, and

- there is no way to verify that a CS has ‘enough’ flexibility, or to discover that ‘more’ flexibility is needed.

It is clear that the notion of flexibility is too general and unspecific to be of value in assessing the quality of a CS design, and does not contribute to an understanding of the evolution of the CS. The main problems with the concept of flexibility are both in the dependence on future events, and its lack of specificity. What is needed is sound criteria that can be measured and researched by looking at the actual schema evolution as changes occur over the operational lifetime, and that can be used to improve current best-practices for CS stability and flexibility. The central goal of this paper is to propose such a set of metrics. We do not claim that the proposed set of metrics is exhaustive but, to the best of our knowledge, the comprehensive set of metrics for schema evolution as defined in this paper have not been reported before in the literature.

The paper is organized as follows. Section 2 introduces the general framework for stability. Section 3 derives the principal requirements for stability and proposes suitable metrics. Section 4 argues the validity of the set of metrics. Section 5 discusses how these metrics can be applied in a field study of schema stability. Section 6 looks at some related work. Section 7 draws conclusions and outlines directions for further research.

2 The Framework

We assume the reader is familiar with the traditional 3-schema architecture [3] (Figure 1). Our interest is in the CS being the single best way of perceiving the Universe of Discourse (UoD), not only at design time but as they both evolve over time. It is in their joint evolution that the CS must demonstrate its stability and flexibility.

Intuitively, flexibility means adaptability, responsiveness to future changes in the environment. And ‘more’ flexibility will mean a smaller impact of change. Stability covers much of the same ground but where flexibility refers to a future capacity for change, stability refers to the past, being evidence that any required changes have been accommodated and that flexibility has been delivered. This leads us to conclude that flexibility and stability share the following three ‘dimensions’ that are orthogonal to each other:

- an environment where changes originate, namely the Universe of Discourse,
- time required to adapt, i.e. the time needed to propagate changes to the other components of the information system, and
- the potential to adapt.

These three dimensions are further refined into a number of high-level mechanisms and best-practices that aid the designer in enhancing the future flexibility of a CS. These mechanisms refine the framework as shown in Figure 2. The large number of mechanisms and their wide variations in scope may possibly explain why there is as yet no generally accepted and unambiguous definition of the concept of schema stability.

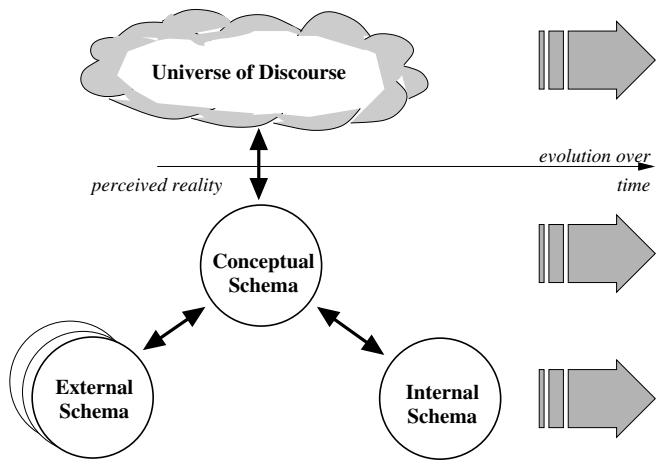


Fig. 1. 3-Schema Architecture

This framework provides a sound basis to evaluate overall flexibility of CSs. It is built on the 3-Schema Architecture and establishes a clear cause-and-effect relationship between ‘structural changes’ in the UoD and those in the CS. It restricts the relevant environment from which changes stem to the Universe of Discourse, and no more. This prevents inappropriate demands of flexibility on the CS. For instance, it excludes changes in responsibilities and tasks of business unit management, changes in the database management system, in the design methodology or duties of the maintenance team etc. An important feature of the framework is that it can be used not only to understand flexibility as a potential for future change. It also provides us with a yardstick to measure to what extent the CS flexibility has actually been exploited in the past. The next section explains the importance of the past evolution of the CS in this respect.

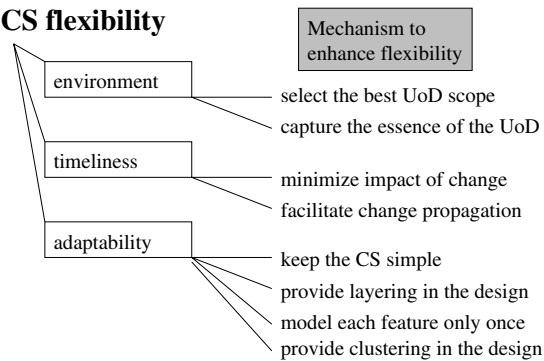


Fig. 2. Conceptual Schema framework for flexibility and stability

This paper is about evolution of the CS over its operational life time. It does not address the issue of quality of the CS as output from the initial life cycle phase of schema design, be it delivered in the traditional ‘waterfall’ method or by some iterative approach. For the same reason, we assume that a single data model theory is used. The data model theory defines the constructs and constructions of the CS, and any change in the data model theory must propagate down to the CS [13]. As a result, changes in the CS are precipitated that are not driven by new user requirements (but by the data management department).

Many design strategies exist that claim to deliver high-quality, flexible CS designs. To name some important ones:

- schema transformations approach [16]
- reflective approaches [40,53]
- global schema integration [3,46]
- component-based development, or: (re)use of schema patterns [10] and
- ontological approaches [49,54]

Why any of these particular strategies should enhance the future flexibility of a CS design is often argued, but the literature is very scarce on actual proof of flexibility in live business cases. While there is no real understanding how these strategies succeed in delivering flexibility, we do not intend to research this issue. The aim of this paper is to understand the mechanisms that are involved in exploiting flexibility as a potential for change.

The life cycle phase of testing, when an unfinished CS is being completed, is also beyond our scope. It is quite common for this phase that many changes occur: be it correction of design failures, or enhancement of initial design quality. But the need for adjustments in this stage indicates progress in the understanding of requirements and improvements in the way of incorporating them into the design. We feel that the amount and types of changes in this phase is a hallmark of the designer’s ability and experience rather than an expression of real changes in the UoD. Some interesting research in this area has been done by [7,8], but not exploring the consequences in the operational life cycle phase.

3 Metrics for Conceptual Schema Evolution

The general framework serves to develop hypotheses on how schema stability ought to be expressed in operational environments. With each hypothesis we associate a metric that may be used to test the hypothesis for evolving conceptual schemas in operational businesses. A metric can be defined as ‘a function whose inputs are elementary measurements of an IT-artifact, and whose output is a single value that can be interpreted as the degree to which the artifact possesses a given property or satisfies a given hypothesis’ [45]. Each of our metrics produces objective (i.e. repeatable) outcomes, and shows the desired tendency for the associated hypothesis.

3.1 Justified Change

By definition, a CS is a complete and correct model of the information structure of the UoD, and nothing else. As long as the business activities of the organization remain unchanged, the information needs remain the same. It follows that a change in the CS is only justified if a change in the UoD information structure is causing it. Any change in the CS that cannot be linked with some driving cause in the UoD is by definition an unjustified change or instability. For instance, the CS should be indifferent to technical changes: increasing transaction volumes, more efficient data fragmentation plans, installation of additional infrastructure etc. So our first demand that must hold in quality CSs is:

Hypothesis:	<i>every change in the CS is justified</i>
--------------------	--

To establish whether a change is justified, we need to

- determine every single CS change, and
- associate each one with the appropriate change driver(s) from the UoD.

The metric for justified change is the ratio of single CS changes that can be associated with an appropriate change driver, over the total number of CS changes (either with, or without change driver). Ideally, the ratio is equal to 1.

The metric is sensitive to the definition of ‘single CS change’. Usually, the ‘single changes’ are identified with elementary, i.e. indecomposable changes as defined in the data model’s taxonomy [6,21]. But care must be taken because many taxonomies consider only transformation of a single construct or construction at a time, while the actual semantics may be a single, coherent change in several schema constructs at once. For instance, dissolving a generalization [55] involves deleting the generalized entity, removing the associated is-a relationships, plus moving all aggregation relationships that the generalization was involved in.

The metric is also sensitive to the demarcation of the UoD. Selecting the right scope for the UoD is an important topic in design and will receive much attention. But once the design phase is finished, the scope of the UoD is fixed. After that, the CS is presumed to be the complete and correct model of the UoD information structure and vice versa: the UoD is that which is modelled by the CS. Consider for instance an enterprise that operates an integrated customer database. To change its CS in order to model which regional offices manage a fragment of the customer database is unjustified because the internal organization of the enterprise has not been included in the UoD. It is suggested in [32] to distinguish between change drivers that are external to the enterprise (“a more stable external environment enhances stability”) and those arising from somewhere within the own organization (“a more simple internal environment enhances stability”).

3.2 Proportional Change

In physics, the property of stability is defined for a system in (near) equilibrium as: any disturbance in the system’s state will cause a reaction that is proportional

to the size of the disturbance. Analogously, we want a small change in the UoD to cause a proportionally small change in the CS, assuming the change is justified. To wit: it is not uncommon that a relatively small change in the UoD triggers an avalanche of changes in the CS. Such a situation deserves to be called an instability, and we want our metrics to single it out as such. So we conjecture

Hypothesis:	<i>every change in the CS will be proportional to the change in the UoD that causes it</i>
--------------------	--

To establish whether a change is proportional to the change driver, we need to measure:

- the size of the change in the CS, and
- the severity of the change driver in the UoD

The metric for proportional change is established as the ratio of size of CS change over the severity of UoD change. Ideally, the ratio should have a low upper bound.

There is a problem here in observing the ‘size’ or ‘severity’ of the single change in the UoD. This concept cannot be formalized rigorously, for the same reason that ‘the information structure of the UoD’ cannot be formalized without referring to some kind of conceptual representation. It is blatantly incorrect to let the maintenance engineer decide on this: the severity of the UoD change will then of course be judged by its impact on the CS! Nevertheless, an operational measure of size could be the number of paragraphs explaining what has changed in the UoD.

In contrast, the size of CS change is easily determined as the number of affected constructs. Depending on the data model theory the size count can be further refined into counts by type such as entity, attribute, constraint etc.

3.3 Proportional Rate of Change

Likewise, it can be said that a system constantly undergoing some kind of change is not very stable. An operational CS which is meant to support many user applications, must have an acceptable low rate of change. But what rates are acceptable, what is not low enough? Users will generally relate it to the business environment that is being modelled. A very turbulent environment changes frequently, and users will accept a correspondingly high rate of change for the CS that models it. That same rate will probably not be accepted in a stable environment, such as a company engaged in the growing of a forest.

Too high a rate is an unstable system, and users and management will not tolerate this for long. On the other hand, a CS with a very low rate of change may not keep abreast with changing business requirements, and might actually be too rigid to change at all. This holds for fragile legacy systems where any change might precipitate an avalanche of unexpected side effects. So we have:

Hypothesis:	<i>the rate of change in the CS will be proportional to rate of change in the UoD</i>
--------------------	---

First, one has to measure the rate of change in the CS. This derives from two measurements:

- the difference between old and new CS, i.e. the number of changes made in creating the new CS version, and
- the lifetime of the CS versions, i.e. elapsed time between subsequent versions going operational.


The rate-of-change is then calculated as the ratio of the number of differences over the version lifetimes. The CS stability expressed in this rate of change improves over time if either the lifetimes of CS versions increase—but this may also reflect rigidity—or if the number of changes between versions decreases.

Next, a measure for rate of change in the UoD must be devised that is targeted at changes in information structure. We are not concerned with changes in information, that are handled by ordinary transactions and data updates. In a similar fashion as above, we propose a rate of UoD measurement to be the ratio of two numbers:

- the difference between old and new user requirements, i.e. the number of changes made in the requirements deriving from the UoD, and
- the lifetime of the consecutive sets of user requirements.

The turbulence in the UoD can then be expressed as the ratio of the number of changes in requirements, over the lifetime of requirements. Of course this is a somewhat hypothetical measurement. When confronted with real business situations, it will be next to impossible to come up with an exact and verifiable ‘count’ of differences in requirements. An alternative is to count the number of change drivers, as discussed above in the metric for justified change.

The metric for proportional rate-of-change is established as the ratio of both measurements: rate-of-change of the CS over rate-of-change in the UoD. Ideally, the ratio should have a low upper bound. A first approximation is to set the lifetime of user requirements equal to the lifetime of the CS versions, making it cancel out of the equation. The metric simplifies to the ratio of:

- the difference between old and new CS, i.e. the number of changes made in creating the new CS version, and
- the difference between old and new  user requirements, i.e. the number of changes made in the requirements deriving from the UoD.

The rate-of-change measurement per CS can be used to benchmark CSs that cover a similar UoD. The CS with lowest rate of change is best, because it will incur lowest cost and least interruption of service to customers. A similar metric was employed by [9] in their study of the evolution of software programs.

There is a caveat, because the rate-of-change measurement is biased. It will appear to be better for small CSs than for highly integrated CSs. If the UoD is larger, then more features of the UoD can change, so the rate of change in the CS will probably be higher. Imagine to cut up a large and complex CS in two parts: the versions for each part can be expected to have half as much changes, and a twice as long lifetime, so the overall rate of change is 4 times as low. The

hypothesis should not be misinterpreted as an advise to fragment large CSs. Other features of a non-conceptual nature may also influence the rate of change: the capacity of the maintenance department. The rate will be significantly lower if the department is understaffed.

This metric is not always applicable. A fundamental assumption is that the entire CS is versioned [43]. Some approaches use other versioning mechanisms, e.g. O-O data modeling theories allow versioning per construct [2]. In such a case the hypothesis may still hold, but the metric, and some others to follow, will not work and another one is needed.

3.4 Compatibility

Compatibility aims to ease change. Compatibility, the demand to keep the impact of change as small as possible, is a natural drive towards stability. It will ease schema evolution because the need for complex data conversions is intensionally minimized.

We define a new CS to be compatible with the old one, if no data present in any construct of the old CS needs to be altered or discarded to fit the new schema. As compatibility will considerably lower overall cost, time and effort of change, designers will go out of their way to achieve it. As a result, a CS change may be compatible, but other quality aspects may be compromised. So we conjecture:

Hypothesis:	<i>the rule is compatible change, the exception is incompatibility at specific places in the schema</i>
--------------------	---

To establish at what locations a CS change is incompatible, we must look at the general pattern of changes in data instances, and ignore for the time being changes in schema constructs. The data that needs attention must be separated from the data that can be left unchanged. By definition, the set of data to be edited is a temporary External View on the old CS. A measure of compatibility for CS change can be based on the relative size of that External View, so we count per type of construct:

- the number of constructs in the ‘data-to-be-edited’ External View, and
- the number of constructs in the old CS

The level of compatibility is then calculated as 1—the ratio of these two counts. Or, equivalently, it is calculated as the number of constructs in the old CS not affected by the change divided by the total number of constructs in the old CS. Ideally, the ratio is equal to 1, when all the data instances of the old schema fit seamlessly in the new schema of things.

Whereas the previous rate-of-change metric was found to be biased towards small CSs, this compatibility metric is biased towards large CSs. If the same change is accomodated in two different CSs in the same way, then the metric produces a more favourable outcome for the larger one.

Compatibility is closely related to the concepts of logical and physical data independence [3,14]. A methodical way for improving compatibility is developed

by [26], but their approach is limited to changes in a single entity (or rather, relation).

Incompatibility is when data instances have to be edited, moved wholly or partially from one entity into another, when relationships have to be reestablished etc. It requires the editing of data instances beyond the scope of either the old and the new CS. Such data conversion efforts are not uncommon in business situations, but are rarely accounted for in the literature [31,33].

A form of incompatibility that is even harder to accommodate is when the level of abstraction changes, causing differences between schemas that are known as semantic discrepancies [47]. A methodical approach that supports the detection and prevention of such incompatibilities in schema evolution is found in [56].

3.5 Extensibility

New ways of doing business are generally supposed to augment existing business procedures and methods, not to replace them. It follows that when information requirements change, the new requirements are additional to what is already accounted for in the old CS. The most obvious changes of this kind are additions of new constructs to the CS.

A type of change that often goes unnoticed is extension of the entity definition. While the entity name and composition are not changed, it is fundamentally altered. This is because the intent is broadened, so many more data instances can and will be recorded for it. An example is when the definition of ‘person’ is first restricted to customers only, whereas after extension it also covers their spouses. A consequence of extension is that the old CS becomes a valid External View on the new CS, preferably an updateable one so that old update routines can remain unchanged. On the data level, change by extension leaves the old data fully compatible with the new schema, as discussed above. This line of reasoning leads us to formulate:

Hypothesis: *the rule is schema extension, the exception is modification of existing constructs*

To establish whether a change in the CS is an extension, we take the metric for compatibility and refine it. For each type of construct in the new CS we count:

- the number of pure additions, and
- the number of constructs in the new CS that differ from the old CS in any way at all

The metric for extension is established as the ratio of the first over the second count. Ideally, the ratio equals 1 meaning that there are only additions and no other changes.

The metric is insensitive to the deletion of constructs, because a deleted construct does not show up in either count. This is unfortunate because a CS

change may appear to be a pure extension while actually, the new construct is a variant of some construction that is deleted simultaneously.

Taxonomies are often based on the idea that change in a construct is a simple concatenation of construct deletion plus construct addition [23]. However, this does not hold at the level of data instances because data will be lost as soon as a construct is deleted. If users are aware that a new construct is actually a variant of something old, then they will demand data compatibility to safeguard their data assets, i.e. that old data instances must be carried over into instances of the new schema. Lossless transformations [16] are introduced into taxonomies to guarantee that any relevant data instances are retained. Therefore the application of metrics for extension and compatibility depends very much on the choice of taxonomy.

3.6 Complexity Hampers Change

It is generally agreed that complexity is a main determinant in maintenance of any product, be it hardware, software, or a conceptual schema [4,15].

As businesses depend more and more on information systems, and as most changes to information systems augment the support for the business operation, it can be expected that the overall size and complexity of information systems will increase.

Surprisingly, the concept of complexity is often discussed only intuitively, for instance [17] introduce their concept of complex object type as ‘simply a boundary line drawn around a set of objects and relationships in the schema’ (p.425).

The usual feeling is that complexity has to do with the combined effect of both a large number of things, and the coupling/interdependence between them, the result being a difficulty to understand the entire setup. The complexity of the composite system is then determined by the number of components, the number of ways in which the components are interrelated, and how these may change over time.

Authors point out that the complexity of a system has a negative impact on its overall quality. As stated by [22] “the more relationships the less comprehension” is possibly due to the accompanying increase in complexity.’ (p.348). We are not interested in complexity as such, but in the effect of complexity of a CS and its constructs on the overall stability. The general idea is that as complexity of a CS is greater, change is more difficult. The maintenance engineer will generally avoid to mess with complex structures, so we conjecture:

Hypothesis:	<i>a more complex CS will change less frequently</i>
--------------------	--

A metric for this hypothesis requires measures for the notions of schema complexity and frequency of change. So if we can decide on objective measures for

- the complexity of each CS version, and
- the lifetime of each CS version

then their ratio is a first characterization for this hypothesis, assuming a linear dependence between the two. Next, the hypothesis should be tested by comparing these ratios for a number of CSs with equal and/or different complexities.

A prerequisite in this hypothesis is the objective indicator of CS complexity—which is hard to find. Size is a first indicator of schema complexity but, as has been observed by [37] ‘this assessment of complexity ignores the number of relationships, named and unnamed, in a given model’ (p.41). Moreover, complexity of a CS is not only dependent on the information structure of the UoD alone. Other factors are of perhaps greater importance, such as ease of use of the data model theory, capabilities of the designer, restrictions due to demand for compatibility etc. [28], when researching software complexity, finds that ‘surprisingly, much of the observed complexity appears to be technically unnecessary (and) excessive schedule pressure and hasty design tend to be a common root cause’ (p.100).

Considering the many aspects that contribute to complexity, it can be doubted that a single number suffices to express overall complexity. For instance, complexity of a CS is very much dependent on the chosen data model theory. Influencing factors are the kinds and levels of abstraction of data model constructs and constructions, the ease of use for maintenance engineers etc. We will briefly discuss two measures for complexity, and their consequences for our metric of change.

A simple measure for complexity of the aggregate mechanism may be obtained by regarding the CS as a lattice where each node is an entity and each edge represents an aggregation relationship. In a more complex lattice, the number of edges (i.e. relationships) will exceed the number of nodes (entities), and integrity constraints are required to ensure overall data consistency. In measuring the complexity of any lattice of a certain size, we need to consider what the ‘minimal’ complexity will be and set this to 0. We also need to account for the fact that some CSs are actually not a single lattice, but are made up of several unconnected subschemas. Our cyclomatic complexity metric for CSs is calculated as:

$$\begin{aligned} & \text{number of unconnected lattices (subschemas)} \\ & - \text{number of nodes (entities)} \\ & + \text{number of edges (relationships)} \end{aligned}$$

A simple lattice like two entities connected by a single relationship has a cyclomatic complexity of 0. Slightly more complex is a lattice of three entities that are all connected, with a cyclomatic complexity of 1. This number has a sound interpretation: it means that 1 constraint may suffice to guarantee referential integrity in the lattice.

Our complexity metric is not new. McCabe’s measure of cyclomatic complexity for software code follows the same line of reasoning; it can even be retraced to the mathematician Euler (1707–1783). [24] apply McCabe’s software metric in 7 case studies in the US Department of Defense. Their findings ‘suggest that maintenance productivity declines with increasing complexity density’ (p.1287), which agrees with our hypothesis. However, closer inspection reveals that the

suggestion actually derives from a single outlier point in their small set of case studies, so the argument is not really strong.

Just like aggregation, the mechanism of generalization can be a cause of complexity. It was noted by [18] how class hierarchies may come to be used inconsistently due to misunderstanding the overall structure of generalizations and specializations.

A measure for complexity for the generalization mechanism is obtained along similar lines. The generalized entity is devolved into a lattice with specializations being nodes, and edges representing the generalization / specialization relationship. Attempts at understanding and clarifying this lattice structure have been described in [11,27,29].

3.7 Abstraction Reduces the Need for Change

The notion of CS abstraction is markedly similar to that of complexity. Like complexity, the level of abstraction is an important design consideration. [15] states that the stability of a CS depends on its level of abstraction. The general idea is that a more abstract design will have a better stability. This is because a less abstract, thus more detailed CS has more constructs and constructions that need to be changed in order to adapt equally well to new requirements. So we conjecture:

Hypothesis:	<i>a more abstract CS will go through less changes</i>
--------------------	--

A metric for this hypothesis should include

- the level of abstraction of the CS, and
- the number of constructs in the CS that change over time

Their ratio is a first characterization for the hypothesis, assuming a linear dependence between the two. In order to test the hypothesis, ratios should be compared for a number of CSs.

Like complexity, the metric for abstraction ought to build on a generally accepted and well-defined measure of abstraction, which again is found to be lacking. It is beyond the scope of this paper to suggest a solution to this issue, but a few remarks are in order. First, it is evident that a CS with a higher level of abstraction should have less constructs with more instances; while a lower level of abstraction results in a CS with more constructs with fewer data instances. Second, abstraction in the CS is strongly related to the data model theory that is used. Some data models (e.g. those based on ontological principles [54]) are considered to be more abstract than others. Third, CS designs are often documented on multiple levels of abstraction [38], and the metric ought to show consistently better outcomes on the higher levels. Finally, it must be noticed that the terms abstraction and clustering (aggregation) are sometimes used interchangeably [22].

3.8 Susceptibility to Change

It is a common assumption that attributes of an entity will change more frequently than the entity as a whole, and descriptive attributes will change more often than primary-key attributes; [1] uses the term ‘sensitivity’. [57] argues that: ‘it is likely that “rules” set by management or other political bodies will change more frequently and quickly than inherent properties, and that rule changes will more frequently affect relationships among entities than the related entities themselves’ (p.1241). In other words, some types of constructs provided by data model theories are presumably more stable than others. Many designers exploit this by doing CS design following the straightforward top-down approach, perhaps calling it abstract-to-concrete. Entities and relationships are presumed to have best stability and hence are modeled first. Attributes and relationship cardinalities are assigned later on, while integrity constraints and business rules are the most volatile and are added to the schema as late as possible. So we conjecture:

Hypothesis: *some types of construct in the CS are more susceptible to change*

Obviously, metrics for this hypothesis must differentiate between the types of construct. A simple measurement will include:

- the various types of construct as provided by the data model theory,
- the total number of constructs per type that is present in the CS, and
- the number of constructs per type that change (perhaps refined by including the type of change, i.e. addition, alteration, or deletion)

The susceptibility to change per type of construct is calculated as the ratio of the number of changed constructs, over their total number in the CS. These ratios can then be compared between types. It seems reasonable to expect that the ratio will be low for entities, while constraints will have a high ratio, meaning they are very susceptible to change. The ratios can also be compared among different CSs.

The hypothesis implicitly assumes that UoD features that are modeled with one type of construct at one time, will be modeled with the same type of constructs at all times. That is, type persistence is assumed [31], while [33] assume a type compatibility invariant when changing a CS. [42] argument that: ‘an object type may not evolve into a method, and a constraint may not evolve into an instance’ (p.357). Some authors concede that a construct might change its type, e.g. in object-orientation [34], but this is not covered by our metric.

We already pointed out that there is no intrinsic reason why type persistence should hold. It is up to the maintenance engineer to decide on the best way to represent UoD features in the new CS, and the choice of construct can differ from the one made in the old CS.

3.9 Preservation of Entity Identity

If the CS is drawn up using a relational data model theory, the previous hypothesis can be applied to changes in the important constructs of candidate-key and functional-dependency. This also sheds some light on the preservation of entity identity, because the set of candidate keys provides a sound understanding of the entity identity as they discriminate each instance of the entity from the others. So we conjecture:

Hypothesis: *the rule is no change in candidate key, the exception is change of composition of keys*

The measurements for susceptibility to change apply, but care must be taken to account for composite keys. What needs to be established is per entity the composition of all candidate keys as present in the old and the new CS, and then determine:

- the number of candidate keys that have been changed from the old CS, and
- the total number of candidate keys for each entity in the new CS.

The ratio of keys changed over the total number of keys is an indication of the susceptibility to change of the candidate keys, and thus of the entity identity itself. If keys are stable, then none will change, and the ratio is equal to 0. It is reasonable to expect that this ratio is tightly linked with the susceptibility-to-change metric for the entities, in other words candidate keys will change only if the entity itself is observed to change.

In a live business environment, it may be very hard to establish beyond doubt what constitutes a change of entity identity. For instance, if an Employee table is defined, do we consider the table intention changed if data on temporary help is entered into the table? A careful count is required that detects homonyms, synonyms and other inconspicuous alterations in the composing attributes; and the count must establish beyond doubt whether any one of the candidate keys is affected by such alterations. □



3.10 Change Is Local

It is a common assumption that changes in the CS are local, i.e. only a single feature of the CS is affected whenever a single requirement changes. As formulated by [5]: ‘every aspect of the requirements appears only once in the schema’ (p.140), or reversely [30] ‘a random grouping of attributes (lack of cohesiveness) will make the E-R model difficult to maintain; however, the database accuracy is not seriously compromised’ (p.685). This aspect of CS stability is often thought to be the result of good schema design. Normalization is generally regarded to take care of this aspect of stability, although normalization targets at eliminating update anomaly in data instances, not in data structures. The assumption being that in a high-quality CS, a single feature of the UoD is modeled in only a single construction of the CS, we stipulate:

Hypothesis:	<i>a single UoD change will cause change in only a single CS construct or construction</i>
--------------------	--

It is evident what should be measured to establish this localization property:

- identify each single change driver in the UoD, and
- determine the number of constructs in the CS that change as a result

The metric is the ratio of the sum of change drivers over the sum of affected constructs. Ideally, this ratio will be equal to 1. Notice that a single CS construct can be affected by two different UoD changes, if that construct is ‘overloaded’ in the sense that it represents more than one UoD requirement.

There is a close relationship with the metric for justified change, but the difference is in the perspective. Justification looks at the changes in the CS and related them to some UoD change driver. The localization metric takes a single UoD change and locates the constructs in the CS that are impacted.

As in the hypothesis of proportional change, there is a problem here as we need to focus on single UoD change. A fairly objective and easy measure of change drivers might be to count the number of paragraphs in the Change Request form, assuming each paragraph identifies a single need for change. A further restriction is that unjustified changes must be ignored for obvious reasons.

3.11 Change Is Restricted to a Single Module

The above claim that changes in the CS are local is often supplemented with a claim that a modular CS has better stability than a CS without modules. The modules are expected to absorb changes and to isolate other modules from the impact of change; comparable to the property of information hiding in O-O approaches. So we conjecture:

Hypothesis:	<i>a single UoD change will cause change in only a single CS module</i>
--------------------	---

We can use the previous measurements and apply them to establish a metric for this localization property after each module and its exact boundaries has been determined:

- identify each single change driver in the UoD, and
- determine the number of modules where a change is made as a result

The metric is the ratio of the sum of change drivers over the sum of affected modules. Ideally, this ratio will be equal to 1 but it may turn out to be higher.

The literature remains vague on the definition and handling of the ‘module’ construct. There is no outstanding best-practice to determine good modules for a CS. Nor can the ‘goodness’ or ‘optimality’ of the chosen modularization be assessed in a rigorous way. Some methods for choosing modules have been described in [17,39,50]. Size (granularity), complexity and even more so the criteria for clustering are critical issues in determining good modules, but it is rarely explained how the right choice will enhance schema stability. It may be speculated

that modularization improves stability by way of the ‘time to adapt’ dimension, because the impact of any change will be confined to only one or two modules.

The exact boundaries between modules are also important, we feel that it is an ‘unjustified change’ if some feature of the UoD is modeled first in one module, but shifted into another one later. Our metric may be included in strategic studies to find out which method may be most favorable in a particular business situation to enhance stability of the CS by modularity.

3.12 Modules Are Stable

Once it is decided to decompose a CS into a set of modules, there will be a feeling that each module has ‘a life of its own’. That is, each module is the valid and complete model of an isolated part of the UoD, and satisfies all the usual quality requirements such as understandability, correctness, data independence etc. The logical implication is that each module can and will evolve as an independent unit within the CS, and its evolution can be traced over time. So we conjecture:

Hypothesis:	<i>modules in the CS are stable</i>
--------------------	-------------------------------------

Some authors take the concept of module even so far that the module is redefined as a single entity [52]. It does have an internal structure, but that remains hidden from outside the module. This is a form of information-hiding, which is a familiar concept in O-O approaches. However, we feel that the idea cannot be easily extended to the relational model, because it infringes upon some of the basic axioms on which the relational data model is built.

Notice furthermore that instability of a module does not mean that the CS as a whole is unstable. The rates of change and levels of complexity and abstraction can also vary greatly among modules, this is related to the dynamics of their corresponding UoDs which may vary from extremely slow to very turbulent. The hypothesis actually brings us back to where we started: to understand stability of the CS. Only now the hypothesis concerns modules only, not the CS as a whole. We gather that all of the previous hypotheses and metrics can be used to study the stability of the CS modules separately.

4 Soundness of the Metrics

Having established the hypotheses on stability and the procedures to measure them by, we must ascertain their quality. Internal validity is: establishing the cause-and-effects as distinguished from spurious relationships. The metrics should produce verifiable outcomes based on clear measurement procedures, and be independent of the observer as well as the timing of the observation. In addition, the metrics ought to show the desired tendencies: a more stable CS should show more favorable outcomes of the metrics, and a less stable CS should show worse metrics. To illustrate this point, consider the example provided by [28] of a careless use of a ‘cost-per-line-of-code’ metric. A more powerful, productive programming language will obviously produce less lines of code. But the cost per

line of code, calculated as the ratio of (variable-cost+fixed-cost) / (lines-of-code) may go up when fixed costs are included in total cost.

We claim that our set of metrics possesses internal validity. This is because they are well associated with our framework for CS stability with three dimensions depicted in Figure 3. This is an argument from theory only; we do not claim validity based on statistical correlations [45]. However, we do not claim that all properties and mechanisms of stability are covered equally well, for instance no metric addresses the “facilitate change propagation” mechanism. This does not signify that the mechanism is unimportant in our framework; if so, it would have been left out. Rather the reason is that any metric for this mechanism involves non-conceptual features of the business environment. The change at the CS level must somehow be sized against the time and effort spent in adapting applications and transaction-processing software, user interfaces, data storage etc. This approach can be seen in project planning methods such as Function Point Analysis, and it is evident that the metrics involved in FPA are not conceptual in nature.

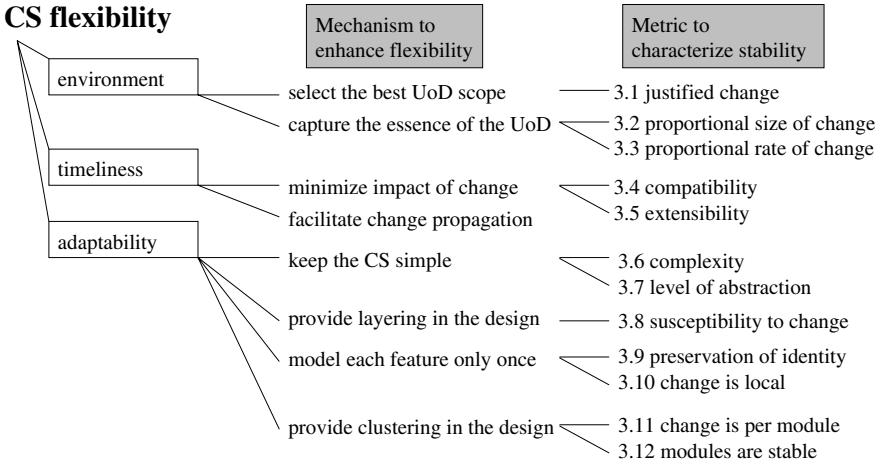


Fig. 3. Metrics to characterize stability based on the framework

Internal validity rests on the fact that the metrics target only conceptual properties of operational CSs. Therefore, all CSs that satisfy the ‘first principles’ of good conceptual composition, can be subjected to our metrics. It can be easily checked that no metric explicitly includes non-conceptual characteristics of the business environment or database system such as:

- overall size of the database, i.e. the same set of metrics can be applied to study small to very large databases
- types of data access, an area covered by CRUD analyses [19], cohesion in methods [4,20] and other approaches
- intensity of data access and volatility (number of daily update transactions)

- number of users and user applications that access the database
- characteristics (constructs and constructions) of the specific Data Model Theory in use
- features of the software- or hardware-architecture of the enterprise such as data distribution or fragmentation across multiple sites, and
- the preferred design approach or the organizational/architectural design strategies.

But there is a complication. The metrics can have an implicit dependence on non-conceptual features of the business environments. Bias was pointed out in several metrics: rate of change, compatibility, and complexity. Finally, it must be kept in mind that the metrics are not geared towards design. If for some UoD, one design approach is superior to all others, given the particulars of the business environment, then this will not be discovered by our metrics.

We have no proof of completeness for our set of metrics, although we consider it rather convincing that the metrics cover the dimensions and mechanisms of the framework rather well. Even so, we cannot claim that all the dimensions and mechanisms of the framework are covered to their full extend. No metric for instance covers the ‘facilitate change propagation’ mechanism. The implication is not that the mechanism is unimportant. If we thought so, it would have been left out. Rather the reason is that any metric for this mechanism must involve non-conceptual features of the business environment.

5 Field Study Setup

Although we claim internal validity of these metrics, we do not claim their external validity. External validity of our set of metrics rests upon their successful application to schema evolution in actual business environments. The setup would be a longitudinal study into the evolution of one, or perhaps several CSs that lie at the heart of vital business information systems. The field study must investigate the phenomenon of change in the CS over a considerable length of time, long enough for the CS to evolve through several versions ([37] requires that at least two CS versions be secured). The aim of the study would be to demonstrate that the metrics can be applied in an operational setup, are objective and reliable enough, that they yield meaningful outcomes, and that they are adequate in understanding the overall flexibility of the CS in the long run.

A first test for feasibility of most of the metrics can be provided by a single case study. Some metrics yield only relative outcomes and to test them would require that multiple business cases be compared. Another argument in favor of multi-case field study is that more testing will lead to outcomes that are more reliable in a statistical sense [45]. Reliability is essential for the next step in our line of research, i.e. to switch from a study of past stability to a prediction of future flexibility. This challenging area of research is to study if, and how our metrics assist in the prediction of future CS changes. Our basic metrics would probably have to be aggregated into more effective ones, in a similar fashion to the approach taken in Function Point Analysis. The field studies can also be

used to test metrics for topics that have not been covered yet, e.g. metrics for derived data, data-dependencies etc.

To detect change in the CS in a business situation can and will encounter problems such as

- lack of documentation. What has been changed in the CS may be discovered with database reverse-engineering methods and tools [44,25], but the business motivation for the change can only be learned from the stakeholders
- abundance of designs that represent the identical semantic structure of the UoD in syntactically different ways
- lack of coordination, where multiple schema releases are being constructed in parallel and the actual sequence of changes implemented in the CS remains uncertain
- strategic changes, such as a switch in the strategies for data processing
- technical change drivers, such as a change of database software. Businesses often find that new software releases invalidates current design decisions, and thus causes serious impact on the existing CS.

All of these problems must be addressed and resolved in order to conduct reliable field study into usability of the metrics. We claim that the study should use operational CSs and be conducted within their business context. The option to use a small-scale experiment is insufficient in our opinion for several reasons:

- real changes in a business environment are always subject to numerous explicit and implicit constraints
- seemingly unrelated changes in other information systems may have an unexpected impact on the present CS
- live systems have a degree of fault tolerance, that allows minor defects to be present in a CS without affecting the overall system quality
- lack of formal CS documentation in legacy systems maintenance, often being balanced by
- huge experience and personal knowledge of maintenance engineers.

We feel that a laboratory setup cannot reproduce these features realistically. To subject metrics intended for an operational environment to empirical validation [4,8] is in our view inadequate.

6 Related Work

Although many data modelling techniques exist that claim to deliver CSs of high quality, relatively few attempts have been made at studying the stability of schemas that are actually produced. It is remarkable that current literature pays so little attention to the important topic of measuring the stability as a determinant of CS quality. For instance, a paper by [36] is devoted to understanding quality in conceptual modeling, but it concentrates on the design phase and mentions modifiability only in a sidebar. Indeed the whole area of software measurement is considered to be immature [12,58].

[33] discusses how a satisfactory schema evolution can be supported. The focus is on enabling the propagation of changes, by creating a series of schema versions that coexist in the database system. However, the taxonomy they use consists of only 9 elementary schema transformations, and only the ‘timeliness’ dimension of CS stability is addressed, ignoring the other dimensions of business environment and schema adaptability.

[37] reports on a research into the stability of 7 CSs denoted in a relational-like data model theory. The following counts are used to define three metrics, namely the ratios of primary, secondary and tertiary attributes over the total number of entities:

- total number of entities
- total number of primary attributes, i.e. those that are essential to understand what the entity represents
- total number of secondary attributes, i.e. relevant data attributes that are not fundamental for understanding
- total number of tertiary attributes, i.e. those used to control and sustain processing needs

The study is limited to a single evolution step of the CSs. It is observed how the average number of primary and secondary attributes per entity increases significantly, whereas the ratio of tertiary attributes per entity halves. We feel that the observations in the report describe symptoms, rather than the essence of the stability problem, and any conclusions drawn from them remain largely intuitive because a formal framework linking the metrics with CS stability criteria is lacking. The idea of the three discerned types of attributes is appealing, but it is unclear what basis it has in theory or accepted best-practices.

[48] describes a longitudinal field study of the evolution in a single relational CS covering parts of both the development period and the operational phase. His findings are that all entities are affected by change at least once over the duration of the field study. However, a serious drawback in his approach is that a very simple taxonomy is used that lacks elements like attribute transformation. It is found that the numbers of attribute deletions and additions are approximately equal, but this finding may indicate that attributes are mostly altered in some way, and this goes undetected because of the poor taxonomy.

[32] develop a theory for strategic information systems planning that includes several hypotheses related to stability and complexity of the systems environment. The theory tries to capture the main determinants for stability and complexity of the strategic information systems, and the tacit assumption is these determinants will also ensure the stability of the CSs that will lie at the heart of the systems.

We have paid little attention to the issue of facilitating change propagation. Database facilities and techniques to enable propagation of changes with minimal interruption of database services are the subject of ongoing research, especially in object-orientation [2,23,41].

Whereas our focus is on evolution in the CS, several researchers are investigating the area of evolving data model theory. Because the impact of changes of data model theory at the CS and data levels can be huge, we feel that such



research should first consider what the intended benefits are for flexibility of the CS and quality of the operational information systems in the long term. Only if proper goals are set can metrics be introduced to investigate and understand what might be called: meta-evolution [51].

7 Conclusions

This paper has introduced a framework for assessing schema stability, consisting of three dimensions. These were further refined into a number of mechanisms and ‘best-practices’ for enhancing the flexibility of conceptual schemas. We used this framework to develop a set of metrics that measure the evolution of the CS with respect to each mechanism. Each metric has been rigorously defined in an operational sense, so that outcomes will be consistent and repeatable when applied to an evolving CS.

Nevertheless, some of our metrics for schema evolution build upon measures for static CS composition, which are not always available and well-defined. For instance, the hypothesis that more abstract CSs will go through less changes, requires a preestablished measure of schema abstraction, which is found to be lacking. But although the metric cannot be defined in an operational way, the hypothesis can still be formulated and indicate the tendency in CS evolution.

The proposed set of metrics, which we do not claim to be exhaustive, can provide valuable insights into the working mechanisms for schema evolution. Only when the elusive relationship between current characteristics of the Conceptual Schema and their behaviour in future changes is well understood, can we hope to improve current practices in database schema evolution.

Research directions. An important goal of current research is to determine stability of an operational CS from a business point of view, i.e. to understand the relationship between the syntactic change in the CS and the semantics of the change driver. To this end, we are validating the metrics as a set of objective measures for stability as a quality aspect of a given CS. Field research is in progress to bring out which of the above metrics are best suited to gauge the stability of schemas, and the impacts of proposed changes. The next challenge in research is to study if, and how our metrics assist in the prediction of future CS changes. We want to use these metrics and develop from them a set of maintenance guidelines how to safeguard and enhance schema quality when faced with changing information requirements and evolving schemas. A related area where research is generally lacking is in bridging the gap between the design and operational phases of the CS life cycle. It is common business experience that the process of mapping a CS into a feasible database schema, requires many implementation choices to be made. A considerable amount of those choices are conceptual in nature, and ought to be incorporated as adjustments and amendments on the CS design. No theoretical framework nor practical research is available that charts the kinds of changes that are made, and whether the effect of changes on the CS is detrimental or beneficial to the schema stability in the long run.

Fundamental research is needed in several areas where clarity of terms is lacking. We already indicated how the notion of schema abstraction needs to be clarified, the same problem was encountered in schema complexity. A promising direction for theoretic as well as applied research is in disclosing the mechanisms underlying our set of hypotheses. This research should include how data model theories contribute to each hypothesis. Proponents of state-of-the-art modelling approaches and design strategies make a variety of claims about schema stability and flexibility. However, their references to stability are mostly unspecific, leaving unclear if claims of stability are substantiated and by what mechanism the promise of stability is realized. A paper is planned to analyse what mechanisms underlies the claims of design strategies, using our framework from Section 2.

Another line of research which can be pursued is strategic alignment, i.e. to match CS stability with business strategy and planning, in order to understand the dynamics of the joint evolution of the business environment and information systems. Other areas where these metrics may prove worthwhile is in estimating cost and effort of a proposed change, in portfolio analysis, in benchmarking organizations on their maintenance performance, etc.

References

1. Amikam A. On the automatic generation of internal schemata, *Information Systems* vol 10 no 1 [1985] 37–45
2. Andany J., Léonard M., Palisser C. Management of Schema Evolution in Databases, 17th Int. Conference on VLDB'91 Very Large Data Bases, Barcelona (Spain), Morgan Kaufmann, San Francisco [1991 09] 161–170
3. ANSI/X3/SPARC Study Group on Data Base Management Systems Interim Report, *ACM SIGMOD Newsletter* vol 7 no 2 [1975 02 08]
4. Basili V.R., Briand L., Melo W.L. A Validation of Object-Oriented Metrics as Quality Indicators, *IEEE Transactions on Software Engineering* vol 22 no 10 [1996 10] 751–761
5. Batini C.W., Ceri S., Navathe S.B. *Conceptual Database Design: An Entity-Relationship Approach*, Benjamin/Cummings Publ. CA [1992]
6. Batini C.W., Di Battista G., Santucci G. Structuring primitives for a Dictionary of ER Data Schemas, *IEEE Transactions on Software Engineering* vol 19 no 4 [1993] 344–365
7. Batra D., Antony S.R. Novice errors in Conceptual Database Design, *European Journal of Information Systems* vol 3 no 1 [1994] 57–69
8. Batra D., Davis J.G. Conceptual data modelling in database design: similarities and differences between expert and novice designers, *Int. Journal of Man-Machine Studies* vol 37 [1992] 83–101
9. Belady L.A., Lehman M.M. A model of large program development, *IBM Systems Journal* vol 15 no 3 [1976] 225–252
10. Castano S., de Antonellis V., Zonta B. Classifying and Reusing Conceptual Schemas, *ER'92 Entity-Relationship Approach*, editors Pernul G., Tjoa A.M., Springer Verlag series LNCS 645 [1992 10] 121–138
11. Chen P.-P., Ming-rui Li The lattice structure of entity sets, *ER'86 Entity-Relationship Approach*, editor Spaccapietra S. Elsevier Science Publ. North-Holland [1986] 217–229

12. Chidamber S.R., Kemerer C.F. A metrics suite for Object-Oriented Design, *IEEE Transactions on Software Engineering* vol 20 no 6 [1994 06] 476–493
13. Claypool K.T., Rundensteiner E.A., Heineman G.T. Evolving the software of a Schema Evolution System, *Database schema Evolution and Meta-Modeling — 9th Int.Workshop on Foundations and Models of Data and Objects (FOM-LADO/DEMM'00)*, editors Balsters H., de Brock B., Conrad S., Springer Verlag series LNCS 2065 [2001] 68–84
14. Date C.J. *An introduction to Database Systems*, volume I, fourth edition, Addison-Wesley Publ. [1986]
15. Delen G.P.A.J., Looijen M. *Beheer van Informatievoorziening*, Cap Gemini Publishing Netherlands (isbn 90-7199650-6) [1992] (in Dutch)
16. de Troyer O. *On Data Schema transformations*, PhD thesis Tilburg University [1993 03]
17. Dittrich K.R., Gotthard W., Lockemann P.C. Complex entities for Engineering Applications, *ER'86 Entity-Relationship Approach 1986* editor Spaccapietra S., Elsevier Science Publ. North-Holland [1986] 421–440
18. Dvorak J. Conceptual Entropy and its Effect on Class Hierarchies, *Computer* [1994] 59–63
19. Ebels E.J., Stegwee R.A. A Multiple Methodology Approach towards Information Architecture Specification, *Proceedings of the '92 IRMA Int.Conference*, Charleston SC, editor Khosrowpour [1992 05] 186–193
20. Etzkorn L., Davis C., Li W. A Practical look at the lack of cohesion in methods metric, *JOOP Journal of OO Programming* vol 11 no 5 [1998 09] 27–34
21. Ewald C.A., Orlowska M.E. A Procedural Approach to Schema Evolution, *CAiSE '93 Advanced Information Systems Engineering*, 5th Int.Conference Paris France, Springer Verlag series LNCS 685 [1993 06] 22–38
22. Feldman P., Miller D. Entity Model clustering: structuring a data model by abstraction, *The Computer Journal* vol 29 no 4 [1986] 348–360
23. Ferrandina F., Meyer T., Zicari R., Ferran G., Madec J. Schema and database evolution in the O2 object database system, *Proceedings of the 21st VLDB'95 Very Large Data Bases conference*, Zürich, editors Dayal U., Gray P.M.D., Nishio S. [1995 09] 170–181
24. Gill G.K., Kemerer C.F. Cyclomatic complexity density and software maintenance productivity, *IEEE Transactions on Software Engineering* vol 17 no 12 [1991] 1284–1288
25. Hainaut J.-L., Engelbert V. DBMAIN: a next-generation meta-CASE, *Information Systems* vol 24 no 2 [1999 04] 99–112
26. Jensen O.G., Böhlen M.H. Evolving Relations, *Database schema Evolution and Meta-Modeling — 9th Int.Workshop on Foundations and Models of Data and Objects (FOM-LADO/DEMM'00)*, editors Balsters H., de Brock B., Conrad S., Springer Verlag series LNCS 2065 [2001] 115–131
27. Jianhua Zhu, Nassif R., Pankaj G., Drew P., Askelid B. Incorporating a model hierarchy into the ER paradigm, *ER'91 Entity-Relationship Approach* [1991] 75–88
28. Jones C. Software Metrics: good, bad, and missing, *Computer* vol 27 no 9 [1994 09] 98–100
29. Jones M.C., Rundensteiner E.A. An Object Model and Algebra for the Implicit Unfolding of Hierarchical Structures [1999] (internet, downloaded on 2000-01-31)
30. Kesh S. Evaluating the quality of Entity Relationship models, *Information & Software Technology* vol 37 [1995] 681–689

31. Lautemann S.-E. A propagation mechanism for populated schema versions, Proceedings of the IEEE Int. Conference on Data Engineering [1997 04] 67–78
32. Lederer A., Salmela H. Towards a theory of strategic information systems planning, J. of Strategic Information Systems vol 5 no 3 [1996 09] 237–253
33. Lerner B.S., Habermann A.N. Beyond schema evolution to database reorganization, Proceedings of the ECOOP/OOPSLA'90 conference, SIGPLAN Notices vol 25 no 10 [1990 10] 67–76
34. Lerner B.S. A Model for Compound Type Changes encountered in Schema Evolution, Technical Report 96-044 Univ Massachusetts [1996 06] (internet, downloaded on 2000.02.29)
35. Levitin A.V., Redman T.C. Quality dimensions of a conceptual view, Information Processing & Management vol 31 no 1 [1995] 81–88
36. Lindland O.I., Sindre G., Sølvsberg A. Understanding Quality in Conceptual Modeling, IEEE Software [1994 03] 42–49
37. Marche S. Measuring the stability of data models, European J. of Information Systems, a publication of the Operational Research Society, vol 2 no 1 [1993] 37–47
38. Mistelbauer H. Datenmodellverdichtung: Vom Projektdatenmodell zur Unternehmens-Datenarchitektur, Wirtschaftsinformatik vol 33 no 4 [1991 08] 289–299 (in german)
39. Pels H.J. Geïntegreerde informatiebanken: modulair ontwerp van het conceptuele schema, Stenfert Kroese Leiden NL [1988] (in dutch)
40. Peters R.J., Tamer Özsu M. Reflection in a Uniform Behaviour Object Model, ER'93 Entity-Relationship Approach 1993, 12th Int. Conference Arlington (Texas), editors Elmasri, Kouramajian, Thalheim, Springer Verlag series LNCS 823 [1993 12] 34–45
41. Peters R.J., Tamer Özsu M. An axiomatic model of dynamic schema evolution in objectbase systems, ACM transactions on Database Systems vol 22 no 1 [1997 03] 75–114
42. Proper H.A., van der Weide T.P. Towards a general theory for the evolution of application domains, Proc. Australasian Database Conference'93, editors Orlowska, Papazoglou, World Scientific [1993 02] 346–362
43. Roddick J.F, Craske N.G., Richards T.J. A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models, ER'93 Entity-Relationship Approach, 12th Int. Conference Arlington (Texas), editors Elmasri, Kouramajian, Thalheim, Springer Verlag series LNCS 823 [1993 12] 137–148
44. Sauter C. Ein Ansatz für das Reverse Engineering relationaler Datenbanken, Wirtschaftsinformatik vol 37 no 3 [1995] 242–250 (in German)
45. Schneidewind N.F. Methodology for Validating Software Metrics, IEEE Transactions on Software Engineering vol 18 no 5 [1992 05] 410–422
46. Shanks G., Darke P. Understanding corporate data models, Information & Management vol 35 no 1 [1999] 19–30
47. Sheth A., Kashyap V., So Far (schematically) Yet So Near (semantically), Proceedings of the IFIP WG2.6 DS-5 Conference, Lorne Australia [1992 11 16] 272–301
48. Sjøberg D. Quantifying Schema Evolution, Information & Software Technology vol 35 no 1 [1993 01] 35–44
49. Storey V.C., Ullrich H., Sundaresan S. An Ontology for Database Design Automation, ER'97 Entity-Relationship Approach 1997, editors Embley, Goldstein, Springer Verlag series LNCS 1331 [1997] 2–15
50. Teorey T.J. Database Modeling & Design: The fundamental Principles, 2nd edition Morgan Kaufmann Publ. [1994]

51. Terrasse M.-N. A Modeling Approach to Meta-Evolution, Database schema Evolution and Meta-Modeling — 9th Int. Workshop on Foundations and Models of Data and Objects (FOMLADO/DEMM'00), editors Balsters H., de Brock B., Conrad S., Springer Verlag series LNCS 2065 [2001] 201–218
52. Urtado C., Oussalah C. Complex entity versioning at two granularity levels, Information Systems vol 23 no 3/4 [1998] 197–216
53. Veldwijk R. Hoe rekbaar is flexibel, Database Magazine [1996] 38–43 (in dutch)
54. Wand Y., Monarchi D.E., Parsons J., Woo C.C. Theoretical foundations for conceptual modelling in information systems development, Decision Support Systems vol 15 [1995] 285–304
55. Wedemeijer L. Semantic Change Patterns in the Conceptual Schema, ECDM'99 Advances in Conceptual Modeling, editors Chen, Embley, Kouloumdjian, Liddle, Roddick, Springer Verlag series LNCS 1727 [1999 11] 122–133
56. Wedemeijer L. A Method to Ease Schema Evolution, IRMA'00 International Conference, Anchorage AK, [2000 05] 423–425
57. Wilmot R.B. Foreign keys decrease adaptability of database designs, Communications of the ACM vol 27 no 12 [1984 12] 1237–1243
58. Zuse H. A Framework of Software Measurement, Walter de Gruyter Berlin, NewYork [1998]

Author Index

Aoumeur, Nasreddine 33

Balko, Sören 61

Barker, Ken 142

Böhlen, Michael H. 115

Chan, Fung-Yee 133

Claypool, Kajal T. 68, 182

Conrad, Stefan 163

Franconi, Enrico 85

Gelbard, Roy 100

Gilmour, Asher 100

Grandi, Fabio 85

Heineman, George T. 68

Jensen, Ole G. 115

Mandreoli, Federica 85

McFadyen, Ron 133

Peters, Randal J. 142

Rundensteiner, Elke A. 68, 182

Saake, Gunter 33, 163

Su, Hong 182

Terrasse, Marie-Noëlle 202

Türker, Can 1, 163

Wedemeijer, Lex 220